Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Matúš Tejiščák

On the semantics of exceptions for high level and low level languages

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: Wouter Swierstra PhD
Study programme: Computer science
Specialization: Theoretical computer science

Prague 2012

I would like to thank to my dear Flu for giving me the key initial push to leave for Nijmegen – thus triggering a sequence of exciting events leading to this work – and for her support all along the way.

Not the smallest bit less, I would like to thank Dr. Wouter Swierstra for his excellent guidance, from the point of helping a confused student pick a topic, through consultations conveying much experience and insight while leaving decisions up to me, to the careful final reviews few days before the submission of this thesis.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, date ..... signature of the author

Název práce: On the semantics of exceptions for high level and low level languages

Autor: Matúš Tejiščák

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: Wouter Swierstra PhD, Software Technology Group of Utrecht University

Abstrakt: V práci se zabýváme korektností kompilátoru jazyka s výjimkami. Předkládáme formální sémantiku; jak denotační sémantiku vysokoúrovňového jazyka, tak operační sémantiku jazyka instrukcí pro zásobníkový stroj. Studujeme metodu odvíjení zásobníku a poté, iterativně ve více krocích, předkládáme modifikovanou metodu. Tato je strukturálně rekurzivní a tudíž vhodná pro implementaci v totálních závisle typovaných jazycích. Nakonec předkládáme implementaci kompilátoru v závisle typovaném jazyce Agda, spolu se strojově ověřitelným důkazem, že předložená implementace kompilátoru při překladu zachovává sémantiku programu.

Klíčová slova: závisle typované programování, výjimky, kompilátor, korektnost

Title: On the semantics of exceptions for high level and low level languages

Author: Matúš Tejiščák

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Wouter Swierstra PhD, Software Technology Group of Utrecht University

Abstract: The thesis deals with correctness of a compiler of a simple language featuring exceptions. We present formal semantics, both denotational semantics of a high-level language and operational semantics of a low-level language for a simple stack machine. We study the method of stack unwinding and then iteratively, improving upon a naive solution, we present a different method that is structurally recursive and thus suitable for implementation in total dependently typed languages. Finally, we provide an implementation of the compiler in the dependently typed functional programming language Agda, along with a mechanically verifiable proof of adherence of the implementation to the semantics.

Keywords: dependently-typed programming, exceptions, compiler, correctness

Contents

Introduction

This thesis deals with the semantics of exceptions in programming languages and how the semantics is preserved by the compilation process.

To verify a compiler that compiles code from a high-level language (such as Haskell) to a low-level language (such as the x86 assembler), we need (besides other things):

- to *state semantics* and properties of the *high-level* language;
- to *state semantics* for *low-level* code that results from the compilation process;
- to *prove* that compilation preserves this semantics;
- to *formalize* the above three points in a way that allows for mechanical verification.

Clearly, without these ingredients, there is no way to define what correctness of a compiler actually means (let alone prove it).

In this thesis, we pursue the above four goals, focused on a simple language with exceptions. Besides these core objectives, we want our solution to have other properties, all related to being useful in practical compiler development:

- the specifications should be *runnable*; that is, the result should be a program that can be run, that can provide compiled code for given input code fragments, and that can execute the compiled code, producing actual results;
- the program should be *readable*. Even though the program may be verified, it should still convey the mechanism of execution clearly without proof clutter getting in the way of comprehension;

- the compiled code should be *executable by a simple machine* (a stack machine, for instance), with fully explicit state and no fancy high-level features such as continuations, arbitrarily-sized instructions or implicit stacks;
- there should be an obvious and *straightforward way to extract* the executable core sans proofs into other (mainstream) languages, either manually or automatically, for practical use.

The contributions of this thesis are:

- We formalize the discussed semantics for a simple language of expressions featuring binary operators[1] and exceptions, using the language as our high-level language (Section 3.1.1);
- we explore approaches to the design of execution of low-level code on a virtual machine, first examining an approach found in the literature, then presenting a modified one that is easier to implement in a dependent setting by gradually improving on a naive solution and explaining the choices (Sections 3.1.2 to 3.5);
- we formalize the semantics of a language of instructions for a simple stack machine, which is our target low-level language, using the latter approach (Section 3.5.3);
- we define a compiler generating instruction sequences from expressions in the high-level language (Section 3.5.2);
- we prove that the semantics is preserved in code generated by this compiler (Section 3.6.3).

We use the general-purpose dependently typed pure functional language Agda in our development. Readers of this thesis are assumed to have knowledge of functional programming, but not necessarily dependently typed programming. A brief introduction to dependent types and related concepts for the purposes of this thesis will be given in Chapter 2.

[1] We actually implement only addition but adding new operators is trivial.

Chapter 1

Exceptions and certified programming

1.1 Exceptions in programming languages

"Exceptions" is a rather broad term referring to a strategy of handling erroneous states in computer programs by interrupting normal program execution, running special code called an *exception handler* and then resuming execution in a known, different state.

The usual terminology is not very strict: the word "exception" may mean slightly different things – or different sides of the same thing – in different contexts. For example, programming languages are said to *have exceptions* if they support this kind of error handling; exceptions are said to *be compiled*, while it is the corresponding infrastructure and support code that is compiled; often the word "exception" denotes a piece of information about the error being handled; et cetera.

When an error occurs during normal execution of a program, *an exception is thrown*[1]. This starts the process of *handling the exception*: looking for a suitable *exception handler* that *handles the exception*, either by *catching* it to resume normal computation, or by *re-throwing* it to find another handler able to deal with the error.

As already mentioned, the word "exception" also denotes a piece of information about the error or condition causing the exceptional state. Exceptions in this sense are simply values of the programming language[2]. These values are provided by the code that throws, handlers can inspect them and behave accordingly.

[1] Some programming languages, such as Python or OCaml, use the term *raised* [RD11].

[2] In many languages, the choice of values that can be thrown is restricted to those specialized for representing exceptions.

Due to common names of the corresponding syntactic features of popular programming languages[3], a piece of code together with attached pieces of handler code is called a *try-block*, and a piece of handler code is called a *catch-block*.

Most languages also provide *finally-blocks*. These are pieces of code attached to a try-block that are guaranteed to be executed after the try-block, whether an exception has been thrown or not. Because of this property, finally-blocks are usually used to clean up resources. In this thesis, we will not model finally-blocks as these can be supplemented by an appropriate use of all-catching exception handlers and they are not too useful in a language without side effects, anyway.[4]

Often, there may be multiple handlers attached to a piece of code, each dealing with a different kind of error. The appropriate handler may be selected by the type of the exception being handled if the language is typed, with most mainstream languages providing support for this approach, or by other means, for example manual inspection of the value thrown.

A try-block needn't have handlers for all exceptions that might arise within. If an exception is *uncaught* within a try-block, it is *propagated* to the containing try-block, which may not catch this exception as well, propagating it further. If an exception propagates all the way out of all nested try-blocks, the program usually aborts.

To give a quick illustration how try-blocks look in the concrete syntax of some widely used languages, Figure 1.1 on the facing page contains four examples.[5]

1.1.1 Purpose

As already mentioned, in practice, exceptions are mostly used to handle errors or other exceptional states. The advantage to using exceptions for this purpose is separation of concerns and hence cleaner resulting code. Especially when reading a program, the reader first reads the code related the expected execution path, uncluttered with error checks, which brings forward the main idea of the code.

However, some languages, for example Python or OCaml, use exceptions also for control-flow purposes, not only in rare and critical events. Python iterators raise an exception to indicate the end of stream [RD11], file-I/O functions in OCaml raise an exception to indicate the end of file [LDF+11], OCaml programmers also sometimes use exceptions to break loops early: for example, upon finding the element sought

[3] Most of them use the same keywords for this purpose.

[4] Finally-blocks may however quickly become useful and considerably non-trivial if combined with mutable state, interrupts, concurrency or other effects.

[5] Note that OCaml does not have syntax for finally-blocks; these are simulated by a function. Haskell does not have syntax for exceptions at all, both *catch* and *finally* are just functions. All code snippets are just symbolic and have been stripped of non-relevant context, such as library imports and the definitions of the functions *perform_work* and *do_cleanup*.

```
try:
        perform_work()
except IOError:
        print "IO error caught"
finally:
        do_cleanup()
```

(a) Python

```
try {
        performWork();
} catch (IOException e) {
        System.out.println(
            "IO error caught");
} finally {
        doCleanup();
}
```

(b) Java

```
performWork
 `catch` (\(e :: IOException) ->
   putStrLn "IO error caught")
 `finally`
   doCleanup
```

(c) Haskell

```
finally do_cleanup (fun () ->
  try perform_work ()
  with IO_error ->
    print_string "IO error caught"
) ()
```

(d) OCaml

Figure 1.1: Try-blocks in different languages

in a list, etc. The implementation of exceptions in these languages is efficient enough to make them cheap and enable this approach.

On the other hand, exceptions may also *cause* subtle errors in programs. The code is less obvious to read since there are no clues at which points the control flow may be diverted because of an exception. Routines/functions have multiple exit points and these may be non-obvious since exceptions may emerge from other parts of the code called within the routine. In languages with manual memory management, resource leaks may occur. Hence correctness of such code is more difficult to assess and reason about by looking at the source code in isolation.

Still, the ubiquity of exceptions in practically all modern programming languages hints at their great usefulness as the standard way of dealing with error states.

1.1.2 History

Exceptions have been present in programming languages for about half a century. Louden and Lambert describe the invention of exceptions in their book *Programming Languages: Principles and Practice* as follows.

> Exception handling was pioneered by the language PL/I in the 1960s and significantly advanced by CLU in the 1970s, with the major design

> questions eventually resolved in the 1980s and early 1990s. Today, virtually all major current languages, including C++, Java, Ada, Python, ML, and Common Lisp (but not C, Scheme or Smalltalk) have built-in exception handling mechanisms. Exception handling has, in particular, been integrated very well into object-oriented mechanisms in Python, Java, and C++, and into functional mechanisms in ML and Common Lisp. Also, languages that do not have built-in mechanisms sometimes have libraries available that provide them, or have other built-in ways of simulating them. [LL12, p. 423]

Over time, various flavors of exceptions and the corresponding infrastructure have evolved in programming languages. Checked exceptions[6] (Java) versus unchecked exceptions (Python) versus optionally checked exceptions (C++); some languages restrict what kinds of values may be thrown as exceptions (Ruby), some do not (C++); some languages do not provide finally-clauses (OCaml), some do not have *any* keywords designated for exceptions at all and use plain functions instead (Haskell), etc.

The variety is large but all these languages share the main principles of exception handling, as described above.

1.2 Certified programming

The rise of dependently typed programming languages brought the opportunity to write *certified programs* in a quite convenient way. Such programs, written in languages with very expressive type systems[7], can include functions whose types represent theorems about the programs themselves, usually about their adherence to specifications. Then, under the Curry-Howard correspondence, implementations of these functions represent proofs [How80]. These proofs are checked by typecheckers mechanically without human intervention.

What's more, such programs needn't ensure their correctness only through pure theorems. They usually also exploit their type systems to assign far stricter types to their ordinary functions, blurring the line between a type signature and a fully specified contract. Programs created in this way can use the type system to enforce invariants in the program, providing a safety net to catch programmer errors, and are often *correct by construction*[8].

[6] A language with checked exceptions requires its functions to specify in their types what exceptions they may throw.

[7] Such as Agda, Coq or Epigram.

[8] In the sense that types are set up to enforce the specification; an incorrect program would not typecheck and subsequently compile.

The programming languages used for these purposes are usually *total* functional languages, meaning that all functions defined in them must be total. A total function terminates on every possible input, always yielding a well-formed value. The trouble with non-termination is that it makes the corresponding logic inconsistent – and this is why totality is a necessary (not sufficient) feature of languages to be used in trustworthy developments.

Certification by a proof of correctness is fundamentally different from testing. A proof gives a *guarantee*, based on the source code of the program in question, that the program follows its specification and never deviates. On the other hand, testing is a stochastic process that, although amenable to coverage improvements, is typically not exhaustible and provides just a good *chance* that if no defects are found, the program is correct.

1.2.1 Certified compilers

However, even when a program is certified, it does not exist in a vacuum. Leroy [Ler09] emphasizes that a program itself is just a text – but correctness of the program is assessed according to some specification, the program needs to be turned into some low-level representation by a compiler and then it needs to be executed on a machine in order to produce results. Hence, if the specification, the compiler, the operating system, the machine, or any other component along the way is incorrect, the actual result can no longer be guaranteed to be correct.

In fact, the impact of bugs in compilers and machines is much greater than that in ordinary applications since all other software depends on them. Yet both compilers and processors are very complex systems performing many non-transparent and complicated tasks.

One way of coping with these issues is using compilers that are certified themselves. For example, a practical compiler for quite a broad subset of the C programming language, called CompCert [Ler09] was developed by Leroy et al. in the Coq proof assistant.

Hence, good specifications and certified compilers provably adhering to them would help making programs more reliable wherever the gains of reliability outweigh the cost of certification. As methods will improve, verification will get cheaper and cheaper; as verification will cover more and more domains, software, compilers, operating systems, hardware, etc., the scope of guarantees – and subsequently gains – will grow.

In this thesis, we will study a specific domain involving specification, compilation and execution of programs with exceptions. We will specify a semantics, create a compiler, prove its adherence to the specification, and examine how this all works together practically in a dependently typed programming language.

Chapter 2

Dependently typed programming in Agda

We will create our development in a dependently typed pure functional language named Agda. This chapter provides a (very) brief introduction to Agda, along with some commentary about related topics.

The aim is to provide the bare necessities for reading this thesis if the reader is not familiar with Agda or dependently typed programming – it is mostly an overview of Agda syntax and some basic techniques used.

For a much more complete introduction to Agda and dependently typed programming, see the tutorial by Norell [Noro8] or any other tutorial from the Agda wiki [Agd12a].

2.1 A simple exceptionless language

As a brief Agda crash-course, we will implement a very simple compiler for a language of arithmetical expressions to instructions for a stack machine.

Being a simple and instructive task, this has been done many times in a variety of programming languages in other papers, publications, blog posts, et cetera; for example in Epigram [MWo6] or in Coq [Chl].

All high-level languages in this thesis will be languages of simple, typed expressions and our first language will feature only natural numbers, addition, and no exceptions. Besides demonstrating how dependent programming in Agda looks like, we will also implement the auxiliary ecosystem of the compiler and the skeleton of our development, upon which we will build later throughout the thesis.

The corresponding Agda code can be found in the attached development, in the subdirectory sec2.1-no-exceptions.

2.1.1 Type universe

It is probably reasonable to expect a programming language to be able to represent expressions of different types (e.g. Integer or Boolean).

Hence the first thing we will introduce is the type universe representing the set of types of expressions of our high-level language. We will *index* our types with values of this type, most notably the type Exp of expressions, to indicate what the type of the expression is.

While we could use Agda types directly for indexing, with an explicit universe, we get decidable equality of expression types[1] and all datatypes conveniently in Set.[2]

NOTE. In this thesis, we will be a bit lax on wording related to type universes. We will be talking about "expressions of the type u", while actually referring to "terms that represent expressions of *the type denoted by u*". This simplified approach can hardly cause confusion, while adhering to precise wording in every situation at all costs would sacrifice comprehensibility.

DATA TYPE DECLARATIONS

Here we arrive at Agda *type declarations*. These resemble GADTs from Haskell or type declarations from other dependently typed languages, like Coq. The first line contains the name of the data type and the Agda universe we want to put the type in. The following lines contain data constructor declarations, including an explicit type for each data constructor.

As a general rule, Agda uses indentation instead of punctuation to delimit blocks. Hence the constructors *must* be indented; in return we get visually clean code.

```
data U : Set where
  nat : U
```

Hence, the above declaration introduces a single constant named nat, that has the type U. This universe could not be much simpler. While we aim to support more types at a later stage, now we restrict the language to express only natural numbers, for the sake of simplicity. As long as we build the program to distinguish different types, this is just a quantitative difference.

Being so simple, this is hardly a representative example of a data type declaration. A bit more elaborate example follows shortly in Section 2.1.2 on page 15.

[1] We do not use this property but we would if we had to implement type-driven handler selection.

[2] To avoid size issues, appearing in Agda in the form of Girard's Paradox [Gir72] (or a simplification thereof by Hurkens [Hur95]), Agda features stratified type universes with universe polymorphism.

Our approach, modeling the types of our expression language as values of the universe U together with an interpretation function, lets us have all types flat in the lowest Agda type universe, called Set, relieving us from the burden of caring about the stratification.

Interpretation function

We will also need an interpretation function that maps types of our simple language to Agda types so that we can use Agda values in the modeled language, talk about its denotational semantics, etc.

Function declarations in Agda consist of a type declaration and a number of clauses describing the outcomes of the function depending on its arguments.

```
el : U → Set
el nat = ℕ
```

There is one clause/equation per distinguished combination of arguments. For example the universe `U` used in a modified version of this function in Section 3.6.1 on page 56 contains another value `bool : U`. The function `el` then contains another clause in the form:

```
el bool = Bool
```

Users of languages belonging (syntactically) to the Miranda family, for example Haskell, Clean or Idris, or even the SML branch of the ML family, will feel safely at home here. The difference here to the other ML-style languages, like OCaml, F# or Coq, is that instead of a big match expression on the right side of a single equation, there are multiple equations where pattern matching takes place on the left side of each equation.[3]

Also note that in Agda, it is possible for a function to return a *type*, in contrast with languages like Haskell or OCaml. This is possible because in dependently typed languages, types, kinds, sorts etc. are all first-class values.

Furthermore, beyond standard type checking, Agda, being a *total* functional programming language, performs two special checks on every function definition.

- The *coverage checker* checks that pattern coverage of every function definition is exhaustive. In other words, whatever arguments are given to the function, *some* pattern from its definition must match them.
- The *termination checker* checks that recursion used (if any) by the given function is structural and hence this function is obviously terminating.

This ensures that every function terminates on every input, which is important for soundness of proofs. If functions were allowed to not terminate, the following circular argument would typecheck as a proof of the apparently false proposition:

```
falsum : ∀ {a : Set} → a
falsum = falsum
```

Put another way, if a function never returns, it can "promise" to return anything, making its type signature meaningless.

[3] Seen for example in the definition of the function `denExp` in Section 2.1.3 on page 17.

2.1.2 Expressions

The core of the high-level language we are going to model consists of its expressions, of course. For now, we will support nothing more than (numeric) literals and addition. However, for further extensibility, we separate the type of binary operators.

OPERATORS

The type family of binary operators is indexed by the types of the two values that the operator accepts as arguments; the third index represents the type of the result of application of the operator on the two values.

```
data Op : U → U → U → Set where
  Plus : Op nat nat nat
```

This means that a value of the type `Op u v w` represents a binary operator whose operands have the types `u` and `w`, and whose result has the type `w`. Hence, the value `Plus` represents an operator that takes two `nat`s and returns a `nat`.[4]

INDUCTIVE TYPE FAMILIES

To put the above type declaration in context from the theoretical point of view, we should briefly discuss various ways how types may be declared in Agda. One of the most elementary kind of a type declaration follows.

```
data Color : Set where
  Red : Color
  Orange : Color
  Green : Color
```

This declaration declares a finite type inhabited by three values represented by the three data constructors `Red`, `Orange`, and `Green`. Such declarations are usually possible even in non-functional languages, often called *enums*.

In Agda however, data constructors may take arguments, as shown in the following snippet.

```
data Observation : Set where
  LightsOn : Bool → Observation
  PersonsPresent : ℕ → Observation
```

Hence we get a type containing values like `LightsOn false` or `PersonsPresent 3`, representing some observations a program might need to model.

What's more, data types can be *recursive*. Instead of the simplest recursive type, the type of natural numbers, let us declare the type of lists of natural numbers.

[4] We will add more operators later. How the constructors representing different operators are typed can be seen in Section 3.6.1 on page 57.

```
data NList : Set where
  [] : NList
  _::_ : ℕ → NList → NList
```

The above declaration says that a list of natural numbers is either the empty list [] or a *cons* (i.e. _::_) of a natural number and a list of natural numbers.

In Agda, the term *inductive types* is used instead of the term *recursive types* for the above kind of declarations[5]. This refers to the totality of the language and finiteness of such data structures, which allows for simple inductive reasoning [BDo8, Sec. 3.1, Remark].

In practice, the declaration of NList is too restrictive because it allows to store only natural numbers in lists. We can lift this restriction by adding a *parameter* to the data type.

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
```

The first line of the declaration says that if the type constructor List is applied to some parameter A being a type, the resulting expression List A is itself a type. This resulting type represents the lists whose elements have the type A, so our original type of natural-number lists is written as List ℕ.

Such types are called *parameterized types* because they abstract out parts of the declaration as parameters that may be chosen arbitrarily[6]. As already mentioned, in the case of lists, the single parameter A denotes the type of the elements contained in a list.

Now suppose we want to write a function head : ∀ {A} → List A → A. We cannot do that, because the list might be empty and there is nothing to return in that case, and Agda does not allow partial functions. We should probably make the function accept only non-empty lists[7].

For that purpose, we need to extend our parameterized type of lists further into an *indexed type family* by adding another argument to the type constructor List[8]. The newly added argument will denote the length of the list.

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)
```

The above code says that the constructor [] constructs a list of any given type with the zero length; the constructor _::_, for any (implicit) length n, takes an element

[5] Agda also provides means to declare and use *coinductive* data types.

[6] Arbitrarily within the given type. In our case, A : Set so the parameter A represents any type.

[7] The other option would be to return Maybe A but sometimes it is not appropriate.

[8] Traditionally, length-indexed lists are called *vectors* so we will also change the name of the constructor to Vec.

of the type A and an appropriately typed list of the length n, yielding a list whose length is suc n.

There are several important points where the newly added *index* differs to the *parameter* A:

- Parameters appear to the left of the semicolon in the type declaration head and they are named because their names stay in scope throughout the whole declaration. Indices appear to the right of the semicolon and they are usually not named. Even if they are, these names are not accessible in the body of the type declaration.

- The parameter A is bound in the declaration head and it is used uniformly within the body of the declaration, whereas the index n in the constructor _::_ is defined locally, only for that particular constructor.

- In the above example, all data constructors must result in the type Vec A n for some value of n. The way the declaration is written, it is not possible to have a constructor construct a value of the type Vec (Maybe A) n.

- Recursive occurrences of a data type must also use the parameters as they were bound in the declaration head. If A is a parameter, the following data constructor declaration is not valid[9]: _::_ : A → List (Maybe A) → List A. Indices are not bound in the declaration head so this restriction wouldn't even make sense for them.

- Therefore, the values of parameters are "chosen from the outside" arbitrarily when the type is used and they parameterize the whole recursive structure uniformly. On the other hand, the values of indices are "chosen by the constructors": they may change in recursive occurrences and different data constructor may construct values of different types, differing in type indices.

  For example, any nonempty length-indexed list contains another list indexed with a different number, smaller by one, but they both store elements of the same type given by the parameter.

- Since type indices depend on the particular constructor used, pattern matching on different constructors yields information about the indices: if we know what constructor was used to construct a value, we also know how the corresponding type indices are related to other values.

[9] In Agda, this is a bit complicated. Type indices are *usually* written to the right of the semicolon in the declaration head. However, Agda does allow using different *parameters* in recursive occurrences of a data type in its constructors. Such "parameters" would be classified as indices by Dybjer [Dyb97], despite being to the left of the semicolon in Agda [Cono8].

> Conversely, this correspondence can also rule out some constructors: if we have a list whose type is indexed by zero, we can be sure that the list was not constructed by the constructor _::_ since zero is not equal to suc n for any n; these expressions do not unify.

In non-dependent functional languages like Haskell or OCaml, (type-) parameterized types are generally available[10]. (Type-) indexed type families are available for example in Haskell in different forms; the form closest to the inductive families of Agda are GADTs [The12, p. 178].

Now that we have declared the length-indexed type family of vectors, we are ready to write the function head that is completely safe and total.

```
head : ∀ {A n} → Vec A (suc n) → A
head (x :: _) = x
```

First, note that the type signature says that the function accepts only nonempty vectors; the length index zero of the empty vector would not unify with the required form suc n.

Second, note that we used only one clause in the definition and the clause for the empty-list case is missing. Agda sees that the type of the argument rules out the constructor [] and does not require us to write a clause for that case.

This concludes our discussion of parameterized types and indexed type families. We will use the above principles in the code that follows.

Expressions

Now we can define the expressions: literals and binary operators. Note that also this type family is indexed with elements of the type universe U so that the Agda type of an expression also incorporates the type of the expression in the modelled language.

```
data Exp : U → Set where
  -- Literals
  Lit : ∀ {u} → el u → Exp u
  -- Binary operators
  Bin : ∀ {u v w} → Bin u v w → Exp u → Exp v → Exp w
```

This is a bit more elaborate declaration using several Agda features.

- Agda supports *Unicode* in its source files and is very liberal about what characters may occur in tokens. This enables Agda source code to get much closer to the standard math notation. Hence the ∀ and → characters are contained in it literally.

[10] Although these too allow definitions like data AltList a b = Nil | Cons a (AltList b a).

What's more, Agda does not classify tokens as operators or identifiers on a lexical basis; (almost) anything may be an infix or mixfix operator, if properly declared[11] and ordinary identifiers may consist purely of special characters. This also means that operators must be surrounded by whitespace: for example, "foo-bar" parses as a single identifier containing a hyphen.

- The names enclosed in braces are *implicit arguments*, as usual in dependently typed languages. For example, a fully saturated application of the constructor Lit contains only one argument; the implicit argument u is inferred from the context of the application.

- The ∀ sign in front of the implicit arguments enables us to *leave out the types* of the arguments – these will be inferred from the context of the constructor declaration. An equivalent declaration of the constructor Lit would be `Lit : {u : U} → el u → Exp u`. This also works with explicit (non-braced) arguments.

- The implicit arguments above are *named*. Explicit arguments can be named, too; we could well write `Lit : ∀ {u} → (x : el u) → Exp u`. However, we do not use the name x anywhere, unlike the name u, so we don't need to even create it.

- Note that we *apply a function in the declaration* of the constructor Lit. In a language with dependent types, we can use function application in type declarations freely.

Given the above definition of the type family of expressions, it can be immediately seen that literals of our expression language always carry appropriately typed values of the type el u.

The typing machinery also ensures that binary operators receive operands of correct types, yielding an expression typed exactly as given by the operator type specification.

2.1.3 Semantics of expressions

Our definition of the high-level language would not be complete without giving the denotational semantics of its expressions. This is done by the following pair of simple functions.

[11] Discussed in Section 2.1.4 on page 18.

Operator semantics

Semantics of operators is given by the function denOp as follows.

```
denOp : ∀ {u v w} → Op u v w → el u → el v → el w
denOp Plus = _+_
```

The function denOp can be interpreted as a function that takes a value representing a binary operator of the type Op u v w and returns an appropriately-typed Agda function of the type (el u → el v → el w).

The function returned for the operator Plus is the ordinary addition function from the standard library. Surrounded with underscores, the infix operator + becomes a standard function identifier.[12]

Expression semantics

Expressions are then turned into Agda values as follows. Literals are evaluated trivially, binary-operator expressions are evaluated using the denotation of the corresponding operator and recursively obtained denotations of the operands.

```
denExp : ∀ {u} → Exp u → el u
denExp (Lit x) = x
denExp (Bin op l r) = denOp op (denExp l) (denExp r)
```

As already mentioned, pattern matching happens on the left side of defining equations; it is *exhaustive*: both of the two possible constructors are covered; and recursion is structural: both recursive applications are made to a subterm of the argument of the function. Hence this function is total.

2.1.4 Virtual machine

We will use a very simple machine to run the compiled code, featuring only a stack of values.

Stack

The stack of the machine is just a cons-list of values, indexed by the list of types (elements of the universe U) of the values pushed on the stack. This means that just by looking at the type of the stack, we can tell how many elements it contains and what types they have. Let us first define the type of stack shapes.

[12] See Section 2.1.4 on the following page for more information on infix operators in Agda.

```
-- open import Data.List
infixr 5 _::_
data List (a : Set) : Set where
  [] : List a
  _::_ : a → List a → List a

Shape : Set
Shape = List U
```

In the definition of the standard list data type above, we encountered a declaration of an infix cons constructor. While it is true that an infix operator surrounded by underscores becomes an ordinary identifier, Agda goes much further and permits (almost) arbitrary prefix, infix, postfix and mixfix operators with an arbitrary number of underscores.

Declaring an identifier containing underscores modifies the Agda parser to recognize occurrences of that identifier where the underscores have been replaced by subexpressions. Combined with Unicode support and the (practical) absence of lexical rules, this is a very powerful device. A few examples (some coming from the standard library) can be seen in Table 2.1.

Mixfix form	*Applicative form*	*Description*
x :: xs	_::_ x xs	standard infix
x !	_! x	postfix factorial
−[x +1]	−[_+1] x	negative whole number constructor
⟨ 2 * x ⟩	⟨_⟩ (2 * x)	non-trivial constructs work fine
x + 1 * y	_+_ x (_*_ 1 y)	fixity and precedence work as defined
if x then y else z	if_then_else_ x y z	mixfix with non-symbols
x ◁ ε	_◁_ x ε	unicode

Table 2.1: Agda mixfix operator examples

The declaration **infixr** 5 _::_ gives fixity and precedence for the operator. The higher the number, the tighter the operator binds. Associativity is then given by the variant of the declaration: **infixr** declares a right-associative operator, **infixl** declares a left-associative operator, **infix** declares a non-associative operator.

Let us return to the compiler. Having defined the type of stack shapes, we can proceed to a definition of the type of stacks.

```
infixr 5 _:-:_
data Stack : Shape → Set where
  snil : Stack []
  _:-:_ : ∀ {u s} → el u → Stack s → Stack (u :: s)
```

The literal snil represents the empty stack; new values are pushed onto it using the infix constructor _:-:_.

Note that pushing a value on the stack changes the type of the stack: the shape index gets prefixed by the type of the value pushed. Given that the empty stack is indexed by the empty shape, we always know how many items there are on the stack and what types they have, as already mentioned above.

Instructions

At this stage, the machine supports only two instructions: PUSH and ADD. This gives rise to the following type family of instructions.

```
data Instr : Shape → Shape → Set where
  PUSH : ∀ {u s} → el u → Instr s (u :: s)
  ADD : ∀ {s} → Instr (nat :: nat :: s) (nat :: s)
```

The type family of instructions is indexed by their action on the stack. The first shape argument is the required stack shape so that the instruction can be executed; the second shape argument is the resulting shape of the stack after the instruction has been executed.

For example, the instruction PUSH takes any value of the type el u and pushes it onto a stack having any shape s, creating a new stack of the shape u :: s.

The instruction ADD represents popping two natural numbers from the stack of any shape with two nats on top of it, hence nat :: nat :: s, and subsequently pushing their sum onto it, resulting in the shape nat :: s.

Code

Finally, code for the stack machine is a sequence of instructions where type indices of subsequent instructions match. For example, if one instruction in the sequence produces a stack of the shape nat :: nat :: s, we want the next instruction in the code sequence to accept this shape.

If we regard Instr : Shape → Shape → Set as a binary relation on Shape, then code is the *transitive reflexive closure* of Instr, which is already included in the Agda standard library as the module Data.Star.

```
-- require import Data.Star
infixr 5 _◃_
data Star {a b : Set} (R : a → b → Set) : a → b → Set where
  ε : ∀ {x} → Star R x x
  _◃_ : ∀ {x y z} → R x y → Star R y z → Star R x z

Code : Shape → Shape → Set
Code = Star Instr
```

The type of instruction sequences is indexed in exactly the same manner as the type of separate instructions: the first index represents the acceptable shape of stack before execution of the piece of code; the second index represents the shape of stack after its execution.

Let us conclude this section with an utility function for concatenation of instruction sequences, which is actually also included in Data.Star.

```
infixr 5 _◅◅_
_◅◅_ : ∀ {R x y z} → Star R x y → Star R y z → Star R x z
ε ◅◅ ys = ys
(x ◅ xs) ◅◅ ys = x ◅ xs ◅◅ ys
```

2.1.5 Execution

Now we will describe how the machine executes instructions, that is, the operational semantics of the low-level language.

At this stage, the state of the machine is fully described by just its stack. This means that there are no other state variables, registers or any additional memory.

INSTRUCTIONS

First, we describe the effects of single instructions on the state of the machine, that is, on the stack, separately.

```
execInstr : ∀ {s t} → Instr s t → Stack s → Stack t
execInstr (PUSH x) st = x :-: st
execInstr ADD (x :-: y :-: st) = (x + y) :-: st
```

The above function simply says that

- the effect of the instruction PUSH is pushing the attached value onto the stack. This consistently extends the information contained in the type of PUSH x.[13] What the type does not say (and execInstr does) is what this value exactly is.
- the effect of the instruction ADD is popping two nats from the top of the stack and pushing their sum back.

Note that in this definition of the execution function, we already reap some benefits of dependently typed programming.

First, of course, Agda checks types of the terms behind the scenes and the machinery of types we have designed so far ensures that in the case for PUSH x, pushing the value x always yields a stack of the desired shape.

[13] The type is Instr s (u :: s) – for some u and s and is interpreted as "PUSH x pushes some value of type u onto the stack".

Second, in the case for ADD, the types ensure that there are always two nats on top of the stack and we can safely pattern-match with the pattern x :-: y :-: st – because this match will always succeed and no other patterns for the ADD case are needed.

Thus the above definition complies to the type signatures involved, which is a relatively solid hint of correctness, and it is *total*, especially no pattern match failures can occur. Despite this, compilers of non-dependently typed languages, like OCaml or Haskell, would complain about non-exhaustive patterns here — there is no way to tell them that, for example, we needn't deal with empty stacks when executing ADD.

CODE

Execution of code is then just a left fold over the sequence of instructions, accepting the initial and yielding the resulting state of the machine.

```
execCode : ∀ {s t} → Code s t → Stack s → Stack t
execCode ε st = st
execCode (i ◁ is) st = execCode is (execInstr i st)
```

Execution of empty code has no effect on the stack; if the code contains instructions, then the first instruction is executed and on the resulting stack, the rest of code is executed.

Note that the type signature of execCode ensures that the code being run modifies (the shape of) the stack consistently with its type indices.

2.1.6 Compiler

Compiling our simple high-level language for a stack machine is easy. The central idea is that execution of an expression of some type is equivalent to pushing its value onto the stack. Literal values are then pushed on the stack directly; binary-operator expressions first evaluate both operands, effectively putting their values on the top of the stack, and then execute the appropriate instruction, determined by the operator. This instruction pops the top two values from the stack as its operands and pushes the result back.

```
-- Syntactic sugar, promote an Instr to singleton Code
⟦ _ ⟧ : ∀ {s t} → Instr s t → Code s t
⟦ i ⟧ = i ◁ ε

-- Determine what instruction performs the required calculation
opInstr : ∀ {u v w} → Op u v w → ∀ {s} → Instr (u :: v :: s) (w :: s)
opInstr Plus = ADD
```

```
-- Turn the expression into code
compile : ∀ {u} → Exp u → ∀ {s} → Code s (u :: s)
compile (Lit x) = ⟦ PUSH x ⟧
compile (Bin op l r) = compile r ⋉ compile l ⋉ ⟦ opInstr op ⟧
```

Again, behind the scenes, Agda ensures that all types match and the code compiled by this function will not make the stack machine fail.[14] For example, there is no way to have the function compile output code where ADD would not get two nats on the top of the stack.

2.1.7 Correctness

This is the only place in this thesis where we include the full proof of correctness. All proofs are of course contained in the attached Agda source code.

AGDA AS A PROOF ASSISTANT

Apart from being a total dependently typed functional *programming* language, the "total dependently typed" part also makes Agda suitable for proving theorems. This means that type signatures can express theorems, values of these types correspond to their proofs, and type checking coincides with proof checking, as already mentioned in Section 1.2 on page 6.

Agda ships with an interactive Emacs editor mode, which extends Agda to a proof assistant and greatly helps writing both proofs and Agda code in general. A (rather brief) guide to the Emacs agda-mode can be found on the Agda wiki [Agd12b].

PROPOSITIONAL EQUALITY

In the following proofs, we will use the operator ≡ to denote *propositional equality*. This is realized in Agda through the following type family.

```
data _≡_ {a : Set} (x : a) : a → Set where
  refl : x ≡ x
```

The definition implies that all nonempty members (inhabited by refl) of this family must have the index the same as the parameter. Therefore conversely, if we have a value refl : a ≡ b, it must be the case that a is the same[15] as b.

[14] To be fair, this is already a property of Code, "inherited" by the function compile via its return type. However, it does constrain possible definitions of the function compile.

[15] The proper term is *definitionally equal*. Agda has an internal notion of definitional equality, based on comparing normal forms.

This is taken into account by Agda when doing pattern matching on the constructor refl and causes unification of the corresponding type variables. Such behavior is not special to propositional equality at all – after all, propositional equality is expressed by an ordinary type family – it is just a consequence of a more general unification mechanism, which works this way for any other data type.

Operator lemma

There are two auxiliary lemmas that we will need to prove our main result. The first one of them is called op-correct and it says that for any binary operator, the instruction picked by the compiler indeed does what the denotation of the binary operator says.

To be more specific, for any operator op and two values x and y of appropriate types, executing opInstr op with the two values on top of the stack results in having the value denOp op x y on the top of the stack afterwards.

```
op-correct : ∀ {s u v w} {st : Stack s} {x : el u} {y : el w}
  → (op : Op u v w)
  → execInstr (opInstr op) (x :-: y :-: st) ≡ denOp op x y :-: st
op-correct Plus = refl
```

In the case for Plus, Agda substitutes the term Plus for the variable op in the appropriate places in the equality, normalizes it (expanding function definitions etc.) and the proof becomes a trivial observation of identity of normal forms, which is indicated by refl.

Distributivity lemma

The other lemma we will need says that execution of code distributes over concatenation of code. In other words, executing the code c ◃◃ d has the same effect as first executing c and then executing d on the resulting stack.

```
compile-distr : ∀ {s t u} {st : Stack s}
  → (c : Code s t) → (d : Code t u)
  → execCode (c ◃◃ d) st ≡ execCode d (execCode c st)
```

We will proceed by induction on the parameter c, which yields two cases: either c is empty or it consists of an instruction and the rest of code. The first case is trivial by substituting ε for the variable c in the equality and observing identity of the normal forms on both sides.

```
compile-distr ε d = refl
```

For writing the proof for the second case, we will use the wonderful way supported by the Agda module ≡-Reasoning[16], which lets us write proofs in the equational-reasoning style; appearing just the way it would if we did it with pen and paper.[17]

```
compile-distr {st} (i ◁ is) d = let open ≡-Reasoning in begin
  execCode (i ◁ is ◁◁ d) st
    ≡⟨ refl ⟩
  execCode (is ◁◁ d) (execInstr i st)
    ≡⟨ compile-distr is d ⟩
  execCode d (execCode is (execInstr i st))
    ≡⟨ refl ⟩
  execCode d (execCode c st)
    □
```

The proof begins with the first line, which is usually exactly the left-hand side of the equality we aim to prove. The second line contains the proof that the first line is equal to the third line and so on – by alternating terms and equality proofs, we can gradually rewrite the left-hand term to the right-hand term of the desired equality.

The first proof is just comparison of normal forms, as indicated by `refl`. In this step, we just unfold the definition of `execCode`, immediately obtaining the next term.

The second proof uses `compile-distr` recursively as the induction hypothesis to break the execution of concatenated code into two stages: first executing `is`, then executing `d`.

The third proof is just `refl` again and we use it to restructure the term to the desired final form, this time *folding* the longer subterm to its definitional equivalent `execCode c st`.

A shorter proof of distributivity

Since `refl` is a certificate of identity of normal forms, by rewriting a term to a different form using `refl` as the proof, the normal form of that term remains the same. As normal forms is what Agda compares when typechecking, we can omit the intermediate `refl`-based rewrites, which leaves us with just one proof in the chain.[18]

```
compile-distr : ∀ {s t u} {st : Stack s}
  → (c : Code s t) → (d : Code t u)
  → execCode (c ◁◁ d) st ≡ execCode d (execCode c st)
compile-distr ε d = refl
compile-distr (i ◁ is) d = compile-distr is d
```

[16] Actually, there are also other similar modules, like ≤-Reasoning etc.

[17] This is achieved by clever mixfix hackery and Unicode usage.

[18] Since a chain consists of equality proofs connected with transitivity of equality, a singleton chain is identical to the single proof itself.

However, often human readability is more desirable than terseness of the code and in such cases, the equational proof may be more appropriate.

Main correctness theorem

This is the central result of this stage that relates together everything we have defined so far in a single proof of correctness.

This proof formalizes the idea that we informally mentioned when we started to write the compiler: executing the compiled code for an expression should be equivalent to pushing the value of the expression (as given by the denotational semantics) onto the stack.

```
correctness : ∀ {u s}
  → (e : Exp u) (st : Stack s)
  → execCode (compile e) st ≡ denExp e :-: st
```

We will proceed by induction on the expression `e`. The literal case is trivial and solvable with `refl`.

```
correctness (Lit x) _ = refl
```

The binary-operator case is a bit more involved and we will prove it using equational reasoning, again.

```
correctness (Binop op l r) st = begin
  execCode (compile (Binop op l r)) st
    ≡⟨ refl ⟩
  execCode (compile r ◃▹ compile l ◃▹ ⟦ opInstr op ⟧ ) st
    ≡⟨ compile-distr (compile r) _ _ ⟩
  execCode (compile l ◃▹ ⟦ opInstr op ⟧ ) (execCode (compile r) st)
    ≡⟨ compile-distr (compile l) _ _ ⟩
  execCode ⟦ opInstr op ⟧ (execCode (compile l) (execCode (compile r) st))
    ≡⟨ cong (λ z → execCode ⟦ opInstr op ⟧ (execCode (compile l) z) (correctness r st) ⟩
  execCode ⟦ opInstr op ⟧ (execCode (compile l) (denExp r :-: st))
    ≡⟨ cong (λ z → execCode ⟦ opInstr op ⟧ z) (correctness l st) ⟩
  execCode ⟦ opInstr op ⟧ (denExp l :-: denExp r :-: st)
    ≡⟨ refl ⟩
  execInstr (opInstr op) (denExp l :-: denExp r :-: st)
    ≡⟨ op-correct op ⟩
  denOp op (denExp l) (denExp r) :-: st
    ≡⟨ refl ⟩
  denExp (Binop op l r) :-: st
    □
```

The first `refl` is used to expand the definition of compile for the `Binop` case so that human readers can see what's going on more easily.

Then we make two appeals to the lemma compile-distr. Each usage of this lemma removes a part of the code sequence (exactly corresponding to an operand of the binary operator being compiled) and transforms it to the effect that this piece of code has on the stack until only a single instruction is left in the code sequence.[19]

The following two rather cryptic steps use the function cong that allows us to prove equality of two terms, given a proof of equality of their subterms in a common context.

```
cong : ∀ {a b : Set} {x y : a}
  → (f : a → b)
  → x ≡ y → f x ≡ f y
```

This function is used with recursive applications of the theorem correctness to both operands of the binary operator. This allows us to rewrite the subterms in the form execCode (compile operand) state to the form denExp operand :-: state, which is equivalent. These two recursive applications are actually inductive hypotheses.

NOTE. In a way, the two steps using exec-distr and correctness for each operand actually correspond to "accelerated execution" of these pieces of code – we do not execute the instructions; instead, we rely on the induction hypothesis to simultaneously remove the code corresponding to the operand and push its denotation onto the stack.

Finally, we use the lemma op-correct to show that executing the leftover instruction is exactly what is left to do to get the desired value on top of the stack.

"WITH" PATTERNS

There is one syntactic feature of Agda left that we have not covered yet but that will occur in the following chapter: the *with-patterns* [Noro8, Section 2.6].

Suppose we wanted to write a function named "first" that, given a (decidable) predicate and a list, returns the first element in the list satisfying the given predicate. We don't want to do it manually; instead, we will use the library function filter that, given a predicate and a list, returns the list of *all* elements satisfying the given predicate.

One way to do it is to define an auxiliary function and compose it with the function filter:

[19] Note that by using underscores instead of proper expressions, we let Agda infer two of three arguments of compile-distr in both applications, which improves readability of the proof.

An underscore in an expression context means "please infer this value" and if the value is uniquely determined from the context, Agda will accept the underscore instead of the (possibly convoluted) inferrable sub-expression.

```
safeHead : {a : Set} → List a → Maybe a
safeHead [] = nothing
safeHead (x :: _) = just x

first : {a : Set} → (a → Bool) → List a → Maybe a
first p = safeHead ∘ filter p
```

Sometimes however, especially with long and complicated type signatures, defining auxiliary functions gets quite inconvenient and these functions, unlike safeHead, are most probably very specific for one purpose only and not reusable. After all, a Haskell programmer might write:

```
first :: (a → Bool) → [a] → Maybe a
first p xs = case filter p xs of
  [] → Nothing
  y:ys → Just y
```

Agda does not have case-expressions because in a dependent setting, pattern matching may yield information. For example, when matching on a proof of equality of variables x and y, the information learned is that these variables should get unified. The core concern is that if such pattern matches occur in case-expressions within a function, the information learned from the case-match would have to influence pattern matching in the arguments of the function.

To deal with this issue, Agda provides *with-patterns*, that effectively add arguments to the function being defined. The usage should be obvious from an example.

```
first : {a : Set} → (a → Bool) → List a → Maybe a
first p xs with filter p xs
first p xs | [] = nothing
first p xs | y :: _ = just y
```

A function can have multiple with-patterns, separated with vertical bars in both the with-clause and pattern-matching clauses themselves.

Finally, notice that we had to write "first p xs" three times redundantly. This is not always the case – sometimes the patterns are more specific in different clauses – but whenever all patterns to the left of the vertical bar are the same, they can be replaced by ellipses:

```
first : {a : Set} → (a → Bool) → List a → Maybe a
first p xs with filter p xs
... | [] = nothing
... | y :: _ = just y
```

The ellipses are understood to stand for the patterns that precede the keyword with in the first clause. Of course, the right sides of the equations can use all variables bound there.

2.1.8 Remarks

Totality of Agda functions gives us a proof of termination of this algorithm and the above correctness proof gives us a guarantee that the compiler calculates the correct code, given the defined semantics.

This is a very strong guarantee and it did not cost us that much – the code we have written looks much like the equivalent in any other functional language. However, we have been maintaining much stronger invariants along the way, being able to, for example, afford including only *relevant*[20] pattern cases in a completely safe way, without triggering compiler warnings.

Implementation-wise, the above sections form separate modules in the accompanying Agda code and these modules define the overall structure of our development. In the following chapter, we will develop the code further by extending and improving particular modules.

[20] In this context, by *relevant* we mean the cases that arise during normal and expected operation of the program; for example, as already mentioned, we needn't specify what to do when the instruction ADD gets an inappropriate number or types of elements on the stack – just because this cannot happen *and the compiler knows it*.

Chapter 3

Compiling exceptions totally correctly

This chapter discusses how the simple language introduced in Chapter 2 can be extended with exceptions.

The title of the chapter correctly suggests that this chapter is loosely based on the paper by Hutton and Wright [HW04], where the basic approach to compiling exceptions was shown. The paper provides excellent insight and inspiration how such code can be written in a language such as Haskell. However, specialties of programming with dependent types, especially termination concerns, will make us diverge in design a bit: the code by Hutton and Wright is not always structurally recursive, which needs to be the case in Agda.

3.1 Compiling exceptions, after Hutton & Wright

The approach by Hutton and Wright was later formalized by Tobias Nipkow [Nip04] in Isabelle, an interactive theorem prover/proof assistant, strictly adhering to the paper by Hutton and Wright; even using the same numbering of lemmas.

In contrast with the original paper and Tobias Nipkow's formalization thereof, our aim is to create a dependently typed program, which means we don't want to *copy* the Haskell code as is and prove it correct; instead, we will *adapt* it for the dependently typed setting and try to expose some appropriate structure.

Furthermore, besides the code performing the actual compilation, we also provide its specification like we did in Chapter 2, including formal semantics of the high-level and low-level languages, and a statement of what it means for an implementation to be correct. We then also provide a machine-checkable proof of correctness of our implementation.

Finally, being implemented in Agda, the specification is both formal and total[1], which explains the "totally" part of the title of this chapter.

The corresponding Agda code can be found in the attached development, in the subdirectory sec3.3-stack-unwinding.

3.1.1 High-level language

EXPRESSIONS

The first module we need to extend when adding exceptions is the one containing the definition of expressions of the high-level language. Namely, we need to add the Throw expression and the Catch construct.

```
data Op : U → U → U → Set where
  Plus : Op nat nat nat

data Exp : U → Set where
  Lit : ∀ {u} → el u → Exp u
  Bin : ∀ {u v w} → Op u v w → Exp u → Exp v → Exp w
  Throw : ∀ {u} → Exp u
  Catch : ∀ {u} → (val : Exp u) → (hnd : Exp u) → Exp u
```

The type of expressions gets two new constructors.

- One of them is Throw, which is similar to the literal constructor Lit, except that no value of the type el u is needed: a throw-expression can promise to yield a value of any type without actually having it.
- The other one is Catch. This constructor takes two expressions of the same type, the regular value and an exception handler, representing a catch-block.

SEMANTICS OF EXPRESSIONS

The expression type has just been extended with two new constructors and we need to formalize what the meaning of the two expression variants actually is. For that purpose, we need to alter the function denExp.

The first change is in the return type of denExp: we need a way to indicate whether the expression evaluates to a value or whether an uncaught exception occurs. To express that, the function denExp will now return Maybe (el u) instead of the more

[1] A total specification covers all possible cases that may occur, that is, all type-correct cases. Note that this does not necessarily mean that the specification is broad; the same can be achieved by using tight type signatures to rule out the cases not sensible enough to be described by the specification, which is what we prefer in this work.

direct el u, using the value nothing[2] to indicate uncaught exceptions and the value just x to indicate that the expression successfully evaluates to the value x.

The second change is adding pattern cases for the newly added constructors of Exp to the denotation function denExp.

```
denExp : ∀ {u} → Exp u → Maybe (el u)
denExp (Lit x) = just x
denExp (Bin op l r) with denExp l | denExp r
⋯ | just x | just y = denOp op x y
⋯ | just _ | nothing = nothing
⋯ | nothing | just _ = nothing
⋯ | nothing | nothing = nothing
denExp Throw = nothing
denExp (Catch e h) with denExp e
⋯ | just x = just x
⋯ | nothing = denExp h
```

We had to alter the original two cases slightly, most importantly the binary operator case, where the result now yields a value (i.e. doesn't throw) if and only if both subexpressions yield values without throwing.

As hinted above, we added a case for the expression Throw: this one never yields a value and always throws; and also case for catch-expressions: if no exception gets thrown in the value, the whole catch-expression is equivalent to the regular value. Otherwise, it is equivalent to the handler value.[3]

Remarks

To keep things simple, the Throw expression does not take a value, unlike its counterparts in most programming languages. In this thesis, we will worry only about whether an exception has been thrown or not, not about its particular value.

It is worth noting that the expression Throw is indexed as an expression that yields a value of *any* type, and the instruction THROW is indexed as if it pushed a value of any type on the stack – without actually having a value of that type (which would of course be impossible if it was the empty type).

This resembles exceptions in typed functional languages; for example the function throw in Haskell:

```
throw :: Exception e ⇒ e → a
```

[2] In contrast with Haskell, Agda uses lowercase initials of the constructors just and nothing.

[3] Especially, if both values throw exceptions, the catch-expression propagates the exception thrown in the handler.

Hence, this constructor is similar to the combinator mplus in Haskell, which combines two possibly failing computations in exactly the same way.

The function throw can "promise" to yield a value of any type because, in fact, it never returns. In a way, it behaves as *bottom*.

Also, in most programming languages, exception handlers can inspect the exceptions being handled and return different values depending on some attributes of the exception. In our language, it would be pointless to do that because our exceptions do not carry values. Furthermore, inspection of exceptions would require us to implement lambdas or a similar mechanism. Thus, our exception handlers are just simple expressions of the same type as the main expression and they have no means to refer to the exception being handled.

This concludes the definition of our high-level language and the rest of this thesis will be mostly devoted to how to make it work operationally.

3.1.2 Virtual machine

What about our virtual stack machine and its low-level language of instructions? What features and instructions do we need to add to make the machine capable of computing with exceptions?

Hutton and Wright give a description of how this can be done [HW04]. Let us extend the machine along the lines drawn by this paper and see how we can adapt their solution to total functional programming with dependent types, having the goals from the Introduction (page 1) in mind.

STACK

First, Hutton and Wright propose altering the type of stacks because besides values, now we are going to push exception handlers on the stack, too.

Unlike Hutton and Wright, we also need to care about stack shapes. The type of stack shapes will no longer be a plain list of types (that is, elements of the universe U); instead, we will distinguish between *values* and *handlers* pushed on the stack. [4]

```
data Item : Set where
  Val : U → Item
  Han : U → Item

Shape : Set
Shape = List Item
```

[4] Actually, there is even more "dependent" approach: we could index the type of stack shapes by the list of handlers on the stack, getting Shape : List U → Set. In this setup, pushing a value on the stack will change its shape but not the type of the shape, while pushing a handler on the stack will change both the shape and the type of the shape. However, we will not take this route as, however promising it may look, it turns out to be more complicated, while the author could not see any advantages this would yield.

A value, denoted by Val u, is an actual value of the type denoted by u; a handler, denoted by Han u, is a piece of code that, when run, leaves a value of the type u on the top of the stack. This naturally leads to the new type of stacks,

```
data Stack : Shape → Set where
  snil : Stack []
  _:-:_ : ∀ {u s} → el u → Stack s → Stack (Val u :: s)
  _!-!_ : ∀ {u s} → Code s (Val u :: s) → Stack s → Stack (Han u :: s)
```

where the constructor :-: corresponds to pushing values and the constructor !-! corresponds to pushing handlers on the stack.

Note that we push arbitrarily large strands of code as single items on the stack, which contradicts one of our design principles – that the code must be executable on a simple stack machine (Introduction, page 1) – and we will finally address this objection in Section 3.5.

Instructions and code

Next, Hutton and Wright introduce three new instructions of the virtual machine: MARK, UNMARK, and THROW. In our code, this change is reflected in extending the Instr type, which must now reside in a mutual block with Code:

```
mutual
  data Instr : Shape → Shape → Set where
    PUSH : ∀ {u s} → el u → Instr s (Val u :: s)
    ADD : ∀ s → Instr (Val nat :: Val nat :: s) (Val nat :: s)
    THROW : ∀ {u s} → Instr s (Val u :: s)
    MARK : ∀ {u s} → Code s (Val u :: s) → Instr s (Han u :: s)
    UNMARK : ∀ {u s} → Instr (Val u :: Han u :: s) (Val u :: s)

  Code : Shape → Shape → Set
  Code = Star Instr
```

The definition of code doesn't change: it is still a simple "index-matching list" of instructions.

Compiler

We will first use the compiler presented by Hutton and Wright in [HW04], studying how execution can be implemented in a dependently typed setting.

The general idea is that before evaluating a (sub-)expression, we push the corresponding exception handler on the stack, if available. After successful execution of the corresponding code, the handler is removed from the stack.

This compiler is quite similar to the one we introduced for simple exceptionless expressions in Section 2.1.6 on page 21. We will keep all definitions: the functions

⟦_⟧ and opInstr, and both cases of the function compile – these will be just extended with two new clauses for the expressions Throw and Catch.

```
compile : ∀ {u s} → Exp u → Code s (Val u :: s)
compile (Lit x) = ⟦ PUSH x ⟧
compile (Bin op l r) = compile r ⋘ compile l ⋘ ⟦ opInstr op ⟧
compile Throw = ⟦ THROW ⟧
compile (Catch e h) = ⟦ MARK (compile h) ⟧ ⋘ compile e ⋘ ⟦ UNMARK ⟧
```

The expression Throw compiles to a single THROW instruction; the expression Catch generates a MARK-UNMARK-delimited block containing the *guarded expression* and the associated handler.

3.2 *Execution: placeholders*

3.2.1 Machine state

The topic of machine state is left implicit in the paper by Hutton and Wright, yet it is one of the most involved parts of this Agda development. We have to make it completely explicit in order to adhere to our objectives.

To put it precisely, the state of the virtual machine includes the following components:

- the position in the executed code ("instruction pointer");
- the stack;
- a bounded number of other flags and variables.

Then, execution is fully specified if for each instruction, we define the effect of that instruction on the machine state.

We will leave the position in the code implicit and we will not count it as a component of the state. The motivation for doing so is the ability to recurse structurally over code, which we would lose if we permitted any manipulation with code (beyond un-consing done by the execution function).

3.2.2 Placeholder method

One approach[5] to execution, not requiring any additional flags and variables, would be extending the stack type with a new constructor; let us call it □:-:_.

[5] This approach is not considered by Hutton and Wright at all; we include it for illustration as the most naïve strategy, whose disadvantages will be gradually solved as we will be switching to better and better approaches.

```
data Stack : Shape → Set where
  snil : Stack []
  _:-:_ : ∀ {u s} → el u → Stack s → Stack (Val u :: s)
  _!-!_ : ∀ {u s} → Code s (Val u :: s) → Stack s → Stack (Han u :: s)
  □:-:_: ∀ {u s} → Stack s → Stack (Val u :: s)
```

The new constructor acts as a placeholder for missing values if an exception is raised.

Note the strong similarity of the new constructor □:-: _ to the THROW instruction and the Throw expression. The instruction THROW is indexed as an instruction that pushes a value of any specified type on the stack – but it actually does not. The Throw expression is indexed as an expression that yields a value of any specified type – but it actually does not. Likewise, the □ constructor is typed in exactly the same way as the constructor that pushes values on the stack – but it does not.

Thus, it is probably not a surprise that the THROW instruction, instead of pushing a value on the stack, pushes the □ placeholder. To be precise, execution would look the following way.

```
mutual
  execInstr : ∀ {s t} → Instr s t → Stack s → Stack t
  -- the original two cases
  execInstr (PUSH x) st = x :-: st
  execInstr ADD (x :-: y :-: st) = (x + y) :-: st
  -- new instructions
  execInstr THROW st = □:-: st
  execInstr (MARK h) st = h !-! st
  -- unmark: no exceptions thrown
  execInstr UNMARK (x :-: h !-! st) = x :-: st
  -- unmark: an exception thrown, handle it by executing the handler
  execInstr UNMARK (□:-:  h !-! st) = execCode h st
  -- miscellaneous exception handling
  execInstr ADD (□:-:  y :-: st) = □:-: st
  execInstr ADD (x :-: □:-: st) = □:-: st
  execInstr ADD (□:-:  □:-: st) = □:-: st

  -- execCode is still a left fold over instructions
  execCode : ∀ {s t} → Code s t → Stack s → Stack t
  execCode ε st = st
  execCode (i ◁ is) st = execCode is (execInstr i st)
```

However, there are multiple downsides to this solution.

First, and most importantly, it's not really a transition to a different low-level language. Instead, it is actually evaluation of the function denExp with an explicit stack. The placeholder □ corresponds to the outcome nothing of denExp, while the regular stack-cons using the constructor :-: corresponds to the outcome just x.

Second, this solution is not elegant in the sense that we need to add exception-handling cases to every instruction (in our case, only ADD). Why should we define how these instructions handle exceptions when they all must do the same thing: just pass the exception forward and do nothing else? Exceptions should be handled by a different mechanism than regular execution.

Third, this approach is also inefficient. Defining how single instructions handle exceptions means that these instructions are going to be executed. However, we can do better: simply skip the appropriate number of instructions – and real machines are good at skipping efficiently.

Fourth, the Agda termination checker rejects the exception-handling case for UNMARK in the function execInstr. Recursion isn't structural here and we need to convince Agda about termination using a decreasing measure or another way.

Fifth, it is not how machines actually work. Apart from doing jumps, real-world exception-handling strategies don't push whole code blobs on the stack. Instead, they linearize exception-handling code with regular code into a single instruction sequence – and then they jump around as needed.

While this low-level representation does work, it leaves much to be desired. In the following sections (and chapters), we will address these objections. Let us begin with the first three of them.

3.3 Execution: stack unwinding

We will adapt the method of *stack unwinding*, described in the paper by Hutton and Wright. The corresponding Agda code can be found in the attached development, in the subdirectory sec3.3-stack-unwinding.

3.3.1 Virtual machine

Now the machine will have two modes of execution, as hinted in [HW04, p. 7].

The normal mode is exactly what one would expect: executing instructions on a stack, nothing surprising.

However, the non-trivial part is the exception-handling mode. When an exception is thrown, two things need to be done in order to execute the exception handler:

- The stack needs to be unwound. This means that items need to be removed from the stack until the first exception handler is found.[6] When this is done,

[6] Here we do not distinguish types of exception handlers so any handler is always appropriate. If we had typed exceptions, we might need to skip handlers that don't match the exception being processed.

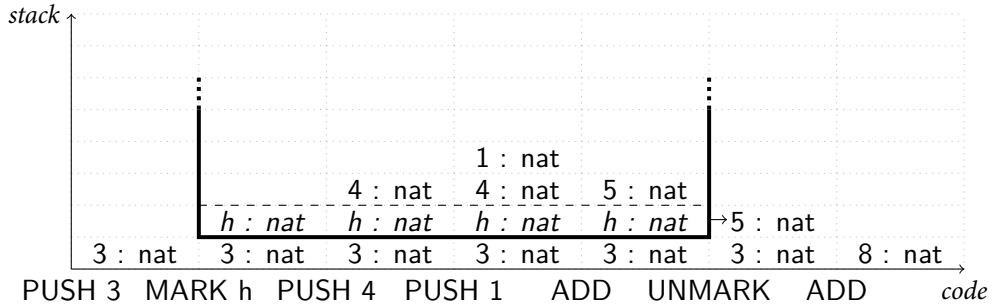

Figure 3.1: Execution of an expression with the handler frame outlined

we can pop the handler from the stack and the resulting stack is now suitable for execution of the popped handler.

- The position in the sequence of instructions needs to be advanced appropriately. In general, we need to skip all instructions that belong to the computation being abandoned.

Handler frames

In a sense, we can talk about *handler frames* that we need to discard while looking for an exception handler. This is best demonstrated on an example. Consider the following expression of the high-level language:

```
Bin Plus
  (Catch
      (Bin Plus (Lit 1) (Lit 4))
      (Lit 2))
  (Lit 3)
```

The above expression compiles to the following instruction sequence, where we use the name `h` for the compiled handler code[7] for readability.

```
[PUSH 3, MARK h, PUSH 4, PUSH 1, ADD, UNMARK, ADD]
```

This code does not throw any exceptions so we already know how it should be executed. Let us plot the intermediate stacks in a graph, with the executed instructions on the horizontal axis, and the stack growing vertically from the baseline. This is how we get Figure 3.1.

The handler frame is delimited by the corresponding `MARK` from the left, the corresponding `UNMARK` from the right, and the corresponding handler on the stack from the bottom. Within the handler frame, evaluation of the guarded[8]

[7] In this case, the handler code is `[PUSH 2]`.

[8] By "guarded" we mean that the expression is the main expression of some catch-expression.

expression runs independently from the context. Finally, after executing UNMARK, a single value is left on top of the stack: this is exactly the denotation of the guarded expression.

Of course, handler frames may be nested since catch-expressions may also be nested arbitrarily. Hence, handler frames are what corresponds to catch-expressions on the operational side of the matter.

STACK UNWINDING

Now that handler frames are defined, we can describe exception handling by stack unwinding quite concisely: abandon the innermost handler frame[9], remembering the handler found at the bottom of the frame, and then execute the handler as ordinary code on the resulting stack.

MACHINE STATE

As for the machine state, we discard our experiments with placeholders and return to the original stack representation with two cons-constructors. Instead of placeholders, we extend the state of the machine by distinguishing between two modes of operation:

- normal operation, where we just need to keep track of the stack;
- exception-handling mode, where we keep additional state variables besides the stack.

3.3.2 Differences to real machines

In real machines, the above distinction would be represented by yet another state variable determining which mode of operation the machine is in. We will model it as different constructors of the State data type.

Like in the placeholder method, we will push handlers on the stack, which contradicts the principles we pursue and isn't really executable by real machines. We will address this issue later, in Section 3.5.[10]

For quitting the current handler frame, we need to skip all instructions belonging to it. This is not so straightforward if we want to keep the recursion structural. The termination checker of Agda requires that our definitions be *obviously* terminating, namely, structurally recursive, which the following definition is not.

[9] Note that this involves discarding stack items but also skipping all instructions that were to be executed within the handler frame.

[10] This also causes trouble with termination but, unlike the executability objection, termination issues can be worked around with standard termination-proving methods or other tricks.

```
execCode : ∀ {s t} → Code s t → State s → State t
execCode nil state = state
execCode (i ◁ is) state = execCode is (execInstr i state)
execCode (THROW ◁ is) state = execInstr (skipToHandler is) state
```

In the snippet above, there is no single argument of execCode that obviously decreases in every recursive call: on the third line, the second argument does not; on the fourth line, the first one does not – and there is no other argument that could possibly take the constantly-decreasing role.

There are several standard ways how to cope with this issue: accessibility predicates, decreasing measures, or rewriting the algorithm to be structurally recursive – and we will aim for the last one.

This is the reason that we "simulate" the jump by switching the machine to an alternative mode of execution and keep the function execCode structurally recursive over the instruction sequence.

Also, the type of the function execInstr, that describes effects of instructions on the machine state, takes only the instruction and state and returns the new state. Since we don't include the "instruction pointer" in the state, there is no way for the instructions to cause jumps in the code in this setup without adding the special states or changing how execInstr works.

3.3.3 Implementation

The normal mode of operation contains simply the stack. However, the exceptional state is a bit more involved; let us declare the data types and the auxiliary functions first and describe them afterwards.

First, we need to know whether there is an appropriate handler on the stack and what type it has. The shape of the stack is sufficient to determine this.

```
-- Get the type of the n-th top-most handler in the Shape.
-- Return nothing if there is no such handler.
unwindHnd : Shape → ℕ → Maybe U
unwindHnd (Han u :: xs) zero = just u
unwindHnd (Han _ :: xs) (suc n) = unwindHnd xs n
unwindHnd (Val _ :: xs) n = unwindHnd xs n
unwindHnd [] _ = nothing
```

We also need to know what shape the unwound stack will have.[11]

[11] The argument order in these functions might be a bit unusual. The reason why the first argument is a Shape, not the number, is that we need to pattern-match on the shape first so that applications of thes functions reduce smoothly in our type signatures.

```
-- Unwind the shape up to just below the n-th top-most handler.
-- Return the empty shape if there is no such handler.
unwindShape : Shape → ℕ → Shape
unwindShape (Han _ :: xs) zero = xs
unwindShape (Han _ :: xs) (suc n) = unwindShape xs n
unwindShape (Val _ :: xs) n = unwindShape xs n
unwindShape [] _ = []
```

Now we can define the data types. We start by defining the type of resumption points, which represent information needed to resume computation after skipping the instructions of the current handler frame.

```
-- Normal operation resumption point.
data Resume (s : Shape) : Maybe U → Set where
  -- A handler is available, also remember the stack on which
  -- the handler should operate.
  Caught : ∀ {u} → Code s (Val u :: s) → Stack s → Resume s (just u)
  -- Uncaught throw.
  Uncaught : Resume s nothing
```

This finally allows us to define the data type of machine states, which represents the two operational modes of the machine: normal mode and exception-handling mode.

```
data State : Shape → Set where
  -- Normal state
  ✓[_] : ∀ {s} → Stack s → State s
  -- Exception-processing state
  ×[_,_] : ∀ {s : Shape}
    → (n : ℕ)
    → Resume (unwindShape s n) (unwindHnd s n)
    → State s
```

As mentioned above, the alternative ✓[_] represents the normal mode of operation, where the machine just needs to keep track of the stack.

The alternative ×[_,_] represents the exception-handling state, described by two (explicit) parameters.

- The parameter n describes how many handler frames we need to *unconditionally* unwind before starting to search for an exception handler.

  This is needed because while skipping instructions belonging to the current handler frame, we might enter additional handler frames nested within. Hence we need to keep track of the depth of nesting.

  For this purpose, Hutton & Wright [HW04, pg. 7] use an implicit stack: in their function skip, the implicit call stack is used to count the nesting levels.

However, our machine does not have a call stack usable for this purpose and we perform instruction skipping by switching to a different machine state so we need to make this counter explicit. Hence we include it as a natural number into the machine state.

- The other parameter describes how to resume normal execution when the machine has finished skipping instructions.

  If there was an appropriate handler on the stack at the time of throwing the exception, this parameter contains the handler and the stack obtained by stack unwinding. If not, the (appropriately typed) constructor Uncaught indicates that an uncaught exception was thrown.

Note that the whole machinery of types ensures a great part of correctness:

- of course, types of all values, handlers and expressions match;
- resumption points representing uncaught exceptions cannot be included in the state if there is a handler on the stack;
- vice versa, resumption points representing handled exceptions cannot be included in the state if there is no handler on the stack.

Also note that in spite of these non-trivial data types used to model the machine state, it is probably obvious that in real implementations, they boil down to just a stack and a couple of additional state variables; with the notable exception of an arbitrarily-sized handler in the case of the constructor Caught, which will be addressed later.

3.3.4 Execution

Now that we know what the state *looks like*, we can take a look at how it *works*. This involves redefining the function execInstr to describe effects of instructions in our new setting.

But first, we need to define a prerequisite for execInstr: the function unwindStack that calculates a resumption record from the given stack. This record is needed for reinstation of computation once all appropriate instructions have been skipped.

```
unwindStack : ∀ {s} → Stack s → (n : ℕ)
  → Resume (unwindShape s n) (unwindHnd s n)
unwindStack (h !-! xs) zero = Caught h xs
unwindStack (h !-! xs) (suc n) = unwindStack xs n
unwindStack (x :-: xs) n = unwindStack xs n
unwindStack snil n = Uncaught
```

The function unwindStack, in accordance with the functions defined above, takes a natural number n denoting how many handler frames are to be discarded right away before starting a search for a handler. If there are no suitable handlers, the alternative Uncaught is returned.

Now we can proceed to the definition of the functions execInstr and execCode, this time in a mutual block.

```
mutual
  execInstr : ∀ {s t} → Instr s t → State s → State t
  -- Normal operation
  execInstr ADD       ✓[ x :-: y :-: st ] = ✓[ (x + y) :-: st ]
  execInstr (PUSH x)  ✓[ st ]             = ✓[ x :-: st ]
  execInstr (MARK h)  ✓[ st ]             = ✓[ h !-! st ]
  execInstr UNMARK    ✓[ x :-: h !-! st ] = ✓[ x :-: st ]
  -- Exception throwing
  execInstr THROW     ✓[ st ] = ×[ zero , unwindStack st zero ]
  -- Nontrivial exception processing
  execInstr (MARK _)  ×[ n     , r ] = ×[ suc n, r ]
  execInstr UNMARK    ×[ suc n , r ] = ×[ n     , r ]
  execInstr UNMARK    ×[ zero  , Caught h st ] = execCode h ✓[ st ]
  -- Trivial exception processing : instruction skipping
  execInstr THROW     ×[ n , r ] = ×[ n , r ]
  execInstr ADD       ×[ n , r ] = ×[ n , r ]
  execInstr (PUSH _)  ×[ n , r ] = ×[ n , r ]

  -- Code execution is still a left fold over instructions.
  execCode : ∀ {s t} → Code s t → State s → State t
  execCode ε st = st
  execCode (i ◁ is) st = execCode is (execInstr i st)
```

The above definition of the function execInstr is mostly straightforward. First, we deal with the normal state, defining how it changes when different instructions are executed.

The first block is essentially equivalent to what we defined for the placeholder method in Section 3.2.2 on page 35.

Then we define what effect the instruction THROW has. The two actions that constitute stack unwinding (Section 3.3.1 on page 36) are represented as follows:

- Popping items from the stack until a handler is found is done by the function unwindStack. If a handler is found, it is returned along with the unwound stack as an instance of the constructor Caught. Otherwise, an instance of the constructor Uncaught is returned. The result of the function unwindStack is then stored in the state of the machine.

- Skipping instructions that belong to the handler frame is done by switching the machine to the instruction-skipping state. The state also contains a natural number that keeps track of nesting depth of handler frames along the way (see the next paragraph for a more detailed description). This value is initially zero.

The reason that we need to keep track of the nesting depth is that instruction skipping always ends at an UNMARK instruction – but not always the first one encountered. The instruction sequence we want to skip may contain more MARK-UNMARK pairs if the current handler frame contains more nested handler frames. Hence we need to count MARKs along the way and then skip that number of UNMARKs before stopping at the real UNMARK we are looking for.

Next, we define the core part of exception processing: dealing with the instructions MARK and UNMARK.

In the exception-processing mode, the effect of the instruction MARK is quite trivial: it just increments the handler frame nesting counter contained in the state.

If the frame nesting counter is nonzero, then the effect of the instruction UNMARK is trivial, too: it just *decrements* the frame nesting counter.

However, if the frame nesting counter is zero, the effect of the instruction UNMARK is a bit more complex: it can be described as switching the machine to the normal mode using the saved stack, and then running the saved exception handler – this is exactly the point where we use the resumption record to reinstate normal operation after having skipped all instructions that were to be skipped.

Note that in the non-trivial case for the instruction UNMARK, we know for sure that the state contains an instance of the Caught constructor[12], so we can be always sure there is a saved handler and stack available for exception handling. The other option is simply ruled out by the type of the stack: we are executing UNMARK, whose type indicates that a handler must be available.[13] This all is understood by Agda and this pattern coverage is accepted as complete.

Finally, we conclude our definition of the function execInstr by defining that in the exception-handling mode, all other instructions are not interesting and they should be simply skipped without having any effect on the machine state.

Note that, unlike in the placeholder method, there is only one case for each such "uninteresting" instruction causing the machine to simply skip it, as opposed to $2^{\text{arity}(\text{instr})} - 1$ cases in the placeholder method, where we had to account for every

[12] As opposed to an instance of the Uncaught constructor.

[13] To be explicit, because the type indices of the instruction UNMARK and of the current state must match, the shape of the current state must contain a value Han u for some u as the second-to-top item. This prevents the function unwindHnd from returning nothing, but since nothing is exactly the index of Uncaught, this constructor cannot occur in this situation.

possible combination of placeholders on the stack. We have reduced both the number of cases, and their complexity.

Also note that in the whole specification, there are no implicit stacks and our functions are completely tail-recursive. We designed the machine to work this way to meet our machine simplicity requirements from the Introduction (page 1).

Both modes of execution are essentially the same; both contain stack, the exception-handling mode also contains a number and a saved exception handler. Thus only the arbitrarily-sized handler violates our simplicity requirement. We will deal with this issue later in Section 3.5.

3.3.5 Termination

However, this solution has a serious flaw: it is not structurally recursive. As already mentioned, Agda will accept only definitions that are *obviously* terminating but the pair of functions execInstr and execCode is not.

The apparent culprit is the call of execCode from within execInstr in the non-trivial, exception-handling case for UNMARK. The function execCode recurses structurally over its first (explicit) argument, the code sequence. However, the handler that is to be executed is not structurally smaller than the code sequence where the UNMARK came from, let alone *obviously* structurally smaller.

There are several ways of coping with this issue, as already mentioned.

First, we can resign on naturally structural recursion altogether and, instead, use the pen-and-paper-like termination proving method: a decreasing measure. This approach is quite straightforward and although it works and offers no surprises, it is tedious and inelegant.

Second, we could exploit a deeper insight in how our high-level and low-level languages are connected and define an intermediate data structure that would facilitate showing correspondence between the two.[14] Such a connection would probably be useful in proving correctness of the compiler, too.

For example, note that since our high-level language is pure, *guarded expressions* are "transactional", in the sense that if execution of any one fails, all traces are cleared up and the handler code is run instead. When considering just effect of the code, it appears that in every handler frame, either the regular code is (completely) executed or the handler code is (completely) executed, which creates a "fork" of possible execution scenarios at the entrypoint of each handler frame – and this can be modeled using a suitable data structure.

However, while it looks promising and the transformation of code to the "fork-

[14] In combination with this approach, the Bove-Capretta method [BC05] can be used to extract termination proof obligations from the code. This way, the termination proof can be separated from the informative part, neither cluttering code with proofs, nor proofs with too much information.

ing" data structure is not too complicated, the author has not been able to prove termination using this approach. This might be an interesting direction for further research.

Third, we can try hard and make the recursion structural. As already hinted, this is the approach we prefer to take in this thesis and we will develop it further in Section 3.4.

3.3.6 Compared to the placeholder method

So what about the objections we raised when evaluating the placeholder method in Section 3.2.2 on page 35?

First, our current low-level language is no longer a trivial model of how the denotation function denExp gets evaluated and its execution is closer to how real machines work.

Second, handling exceptions uses a different mechanism than actually executing every instruction. In the exception-handling mode, (non-interesting) instructions are skipped without much fuss: there is only one case per instruction, compared to 2^{arity} in the placeholder case.[15]

Third, actual efficiency of the pattern match is comparable to that of the placeholder method. We actually cannot do better than going through every single instruction if we want to recurse structurally over code. However, the work we do at every instruction[16] in the exception-handling mode is trivial: we simply skip it, doing no stack inspection, no manipulation with machine state at all. Hence, when entering the exception handling mode, the machine focuses just on the code sequence and starts to search an appropriate place at which to resume execution. This approximates real-world jumps better although we still need to keep track of what instruction is being skipped and when we need to stop skipping.

Fourth, unfortunately, our recursion is still not structural and the termination checker rejects the recursive call to execCode in the handler-running clause of execInstr. However, we are going to address this shortcoming soon in Section 3.4.

Fifth, we still keep pushing blobs of code on the stack so we haven't improved this aspect yet – but we are going to as well, in Section 3.5.

[15] Theoretically, we could do even better and replace all trivial cases with _, the wildcard pattern. However, if we do that, Agda will complain because it cannot check that the types involved are correct – they are slightly different across different cases and they need to be checked separately.

[16] Well, every one except MARK and UNMARK.

3.3.7 Correctness

We don't prove correctness of this implementation because we don't have a proof of termination and since we are going to fix that soon, let us leave it that way.

3.4 Execution: handlers at UNMARK

Our (non-)termination trouble arises from the fact that before running an exception handler, we move it around, push it on the stack, run other code, pop it from the stack etc., which obscures the fact that we never execute the same instruction twice.

Instead of partitioning the code at runtime, shuffling the partitions and executing them in a different order, it is better to already *generate* the code in a more convenient way.

A surprisingly simple change solves all the terminating trouble: let us just attach handler code to the instruction UNMARK instead of the corresponding MARK.

The corresponding Agda code can be found in the attached development, in the subdirectory sec3.4-handlers-at-unmark.

3.4.1 Virtual machine

First, we alter the definition of the type of instructions from Section 3.1.2 on page 33 a bit – we make the UNMARK constructor take the exception handler instead of the MARK constructor.

```
data Instr : Shape → Shape → Set where
  PUSH : ∀ {u s} → el u → Instr s (Val u :: s)
  ADD : ∀ {s} → Instr (Val nat :: Val nat :: s) (Val nat :: s)
  MARK : ∀ {u s} → Instr s (Han u :: s)
  UNMARK : ∀ {u s} → Code s (Val u :: s)
    → Instr (Val u :: Han u :: s) (Val u :: s)
  THROW : ∀ {u s} → Instr s (Val u :: s)
```

Second, we also need to alter the type of stacks from Section 3.1.2 on page 33 because we will no longer push handlers on the stack. However, we want to preserve the overall principle of execution so we have to push *something* to keep the stack well-typed. Hence, we will push a placeholder[17] instead of code, indicating that a handler can be found at the appropriate UNMARK.

[17] Note that this placeholder is different from the one used in the placeholder execution method, where we pushed bogus *values*, not handlers.

```
infixr 5 _:-:_
infixr 5 □!-!_
data Stack : Shape → Set where
  snil : Stack []
  _:-:_ : ∀ {u s} → el u → Stack s → Stack (Val u :: s)
  □!-!_: ∀ {u s} → Stack s → Stack (Han u :: s)
```

3.4.2 Compiler

We will make just the small obvious change in the compiler from Section 3.1.2 on page 34: move the compiled handler from MARK to UNMARK.

```
compile : ∀ {u s} → Exp u → Code s (Val u :: s)
compile (Lit x) = ⟦ PUSH x ⟧
compile (Bin op l r) = compile r ⋘ compile l ⋘ ⟦ opInstr op ⟧
compile Throw = ⟦ THROW ⟧
compile (Catch e h) = ⟦ MARK ⟧ ⋘ compile e ⋘ ⟦ UNMARK (compile h) ⟧
```

3.4.3 Machine state

With this approach, machine state is simplified quite a lot. There are no resumption records and both execution modes contain just the stack (augmented with a nesting counter in the exception-handling case).

```
data State (s : Shape) : Set where
  ✓[_] : Stack s → Shape s
  ×[_,_] : (n : ℕ) → Stack (unwindShape s n) → State s
```

3.4.4 Execution

Execution is simplified as well. Suddenly, recursion is naturally completely structural. In the code, seen in Figure 3.2 on the next page, we will use the auxiliary functions unwindShape (Section 3.3.3 on page 40) and unwindStack (Section 3.3.4 on page 41), trivially modified to work with the stack type we are using here.

Note that:

- the ×[_] constructor now takes only the (unwound) stack, nothing more;
- in the third and fourth clause of the function execInstr, we push and pop only placeholders instead of exception handlers;

```
mutual
  execInstr : ∀ {s t} → Instr s t → State s → State t
  -- Normal operation
  execInstr ADD         ✓[ x :-: y :-: st ] = ✓[ (x + y) :-: st ]
  execInstr (PUSH x)    ✓[ st ]             = ✓[ x :-: st ]
  execInstr MARK        ✓[ st ]             = ✓[ □!-! st ]
  execInstr (UNMARK _)  ✓[ x :-: □!-! st ]  = ✓[ x :-: st ]
  -- Exception throwing
  execInstr THROW       ✓[ st ] = ×[ zero , unwindStack st zero ]
  -- Nontrivial exception processing
  execInstr MARK        ×[ n      , st ] = ×[ suc n, st ]
  execInstr (UNMARK _)  ×[ suc n , st ] = ×[ n     , st ]
  execInstr (UNMARK h)  ×[ zero   , st ] = execCode h ✓[ st ]
  -- Trivial exception processing : instruction skipping
  execInstr THROW   ×[ n , st ] = ×[ n , st ]
  execInstr ADD     ×[ n , st ] = ×[ n , st ]
  execInstr (PUSH _) ×[ n , st ] = ×[ n , st ]

  -- Code execution is still a left fold over instructions.
  execCode : ∀ {s t} → Code s t → State s → State t
  execCode ε st = st
  execCode (i ◁ is) st = execCode is (execInstr i st)
```

Figure 3.2: Handlers at UNMARK: execution functions

- in the eighth clause, we call execCode recursively; this time however, the code argument h is *structurally smaller* than that of the (grand-)parent call to execCode. Agda recognizes this and accepts our definition without complaining.

This is actually all it takes to make execution work with this approach.

3.4.5 Correctness

Since we have eliminated the termination trouble, we can proceed to proving correctness. We do this by induction on the given expression, in a way very similar to how the proof was done in Section 2.1.7 on page 25, using equational reasoning.

The complete proof can be found in the attached code; here we will just outline the main theorem and the accompanying lemmas.

The theorem has almost the same shape as before in Section 2.1.7 on page 25, namely:

```
correctness : ∀ {u} (e : Exp u) (s : Shape) (st : State s)
  → execCode (compile e) st ≡ (denExp e ::: st)
```

This time, we don't prove the correctness with any *stack* but with any *state*, which is more useful in the proof (especially when using the induction hypothesis).

The operator ::: is the "smart stack pusher" that pushes values to the stack contained within a state, dealing correctly with all combinations of cases that may occur; for example, the denotation function denExp may return nothing and the state st may be in the process of handling an exception.

```
infixr 5 _:::_
_:::_ : ∀ {u s} → Maybe (el u) → State s → State (Val u :: s)
just x    ::: ✓[ st ]      = ✓[ x :-: st ]
nothing   ::: ✓[ st ]      = ×[ zero , unwindStack st zero ]
just x    ::: ×[ n , st ]  = ×[ n , st ]
nothing   ::: ×[ n , st ]  = ×[ n , st ]
```

The use of this operator makes our theorems quite neat and concise.

As already mentioned, we prove this correctness theorem by induction on the expression e, using the following three lemmas.

The first lemma we use is the distributivity lemma; again, practically identical to that in Section 2.1.7 on page 23.

```
distr : ∀ {s t u} (st : State s) (c : Code s t) (d : Code t u)
  → execCode (c ◃◃ d) st ≡ (execCode d ∘ execCode c) st
```

We also use two lemmas that perform specialized case analysis. When doing induction on the structure of the expression while proving correctness, there are two

non-trivial cases: the case of binary operators[18] and the case of catch-expressions.[19] Apart from containing sub-expressions recursively (thus requiring the induction hypothesis), it is non-trivial to show that the effect of the code compiled from these expressions conforms to their denotational semantics with respect to exception propagation. This is where we use the two lemmas to analyze all these cases, one for the constructor Bin, the other for the constructor Catch.

```
lemma-op : ∀ {s t u v} (r : Exp t) (l : Exp u) (op : Op u t v) (st : State s)
  → execInstr (opInstr op) (denExp l ::: denExp r ::: st)
    ≡ denExp (Bin op l r) ::: st

lemma-catch : ∀ {s u} (e : Exp u) (h : Exp u) (st : State s)
  → (∀ {s} (st' : State s) → execCode (compile h) st' ≡ denExp h ::: st')
  → execInstr (UNMARK (compile h)) (denExp e ::: execInstr MARK st)
    ≡ denExp (Catch e h) ::: st
```

The function lemma-catch also takes the induction hypothesis as an argument, instantiated for the compiled handler code.

Both functions are quite simple – they consist of 12 to 16 clauses performing just case analysis on the given arguments and with-patterns. All clauses are trivial: they have a simple refl on the right-hand side.

This is all we need to prove correctness (as stated in Section 3.4.5 on the previous page): the cases for the constructors Throw and Lit can be trivially discharged using refl; the cases for the constructors Bin and Catch are proved equationally, using the above lemmas.

3.4.6 Remarks

We have solved the fourth objection introduced in Section 3.2.2 on page 35: recursion is now structural, termination follows trivially from this fact and need not be proved explicitly.

What's left is the fifth problem related to pushing code blobs on the stack. This is solved by linearizing handlers in the main code sequence and adding jumps so that these handlers are skipped unless actually needed.

[18] The constructor Bin.
[19] The constructor Catch.

3.5 Linearized code

As already mentioned, if we linearize handler code within the main instruction sequence, we need jumps for skipping handlers that are not needed. Since, due to structurality requirements on recursion, we cannot simply jump in the code, we need to make several changes throughout our development.

The key difference to the previous section lies in how code for exception handlers is laid out. The overall idea remains the same; however, compared to the compiler in Section 3.4.2 on page 47, we change the layout for catch-expressions.

The instruction UNMARK no longer takes any piece of code. The role of this instruction is taken by the newly-introduced instruction HANDLE, which introduces the handler code. The instruction UNMARK now indicates the end of the handler,[20] as sketched in Figure 3.3.

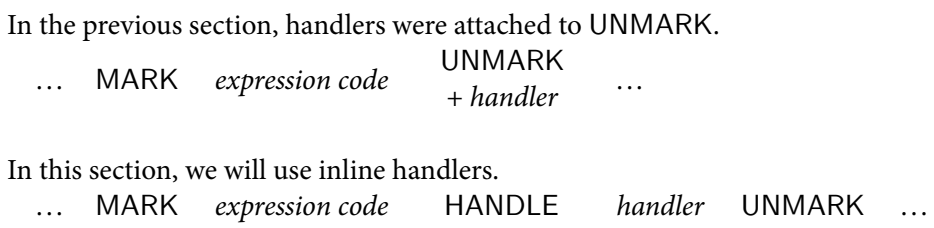

Figure 3.3: Linearized code: layout

The corresponding Agda code can be found in the attached development, in the subdirectory sec3.5-linearized-code.

3.5.1 Virtual machine

The first component we change is the virtual machine, where rather small changes affect almost every aspect. Details of machine state will be covered a bit later in Section 3.5.3 on page 53.

Stack

This code layout leads to changes in the shape of stacks and the stack itself.

Besides handler placeholders, we will push yet another kind of placeholders on the stack, which leads to addition of another item constructor:

[20] This *change* of naming may be a bit confusing with respect to our previous development but the *result* is more logical and intuitive: MARK-HANDLE-UNMARK.

```
data Item : Set where
  Val : U → Item
  Han : U → Item
  Skp : U → Item

Shape : Set
Shape = List Item
```

These new placeholders will not actually carry any information; their purpose is to keep the stack well-typed during skipping. The resulting stack type now changes to:

```
infixr 5 _:-:_ han:-:_ skp:-:_
data Stack : Set where
  snil : Stack []
  _:-:_ : ∀ {u s} → el u → Stack s → Stack (Val u :: s)
  han:-:_ : ∀ {s} → {u : U} → Stack s → Stack (Han u :: s)
  skp:-:_ : ∀ {s} → {u : U} → Stack s → Stack (Skp u :: s)
```

Besides values, we will push two kinds of placeholders on the stack: the placeholder `han u` to indicate availability of a (complete) handler of the type `u`, and the placeholder `skp u` to indicate that a (not necessarily complete) handler of the type `u` is currently present in the code sequence and may be skipped.

If a handler is present in the code sequence completely, both corresponding placeholders occur in the stack. If the handler is being skipped or executed, just the `skp` placeholder occurs in the stack.

INSTRUCTIONS AND CODE

The type of instructions and code needs to be altered as well, due to the above changes.

We rename the instruction UNMARK to HANDLE, adding a new UNMARK with a different meaning.

```
data Instr : Shape → Shape → Set where
  PUSH : ∀ {u s} → el u → Instr s (Val u :: s)
  THROW : ∀ {u s} → Instr s (Val u :: s)
  ADD : ∀ {s} → Instr (Val nat :: Val nat :: s) (Val nat :: s)
  MARK : ∀ {u s} → Instr s (Han u :: Skp u :: s)
  HANDLE : ∀ {u s} → Instr (Val u :: Han u :: Skp u :: s) (Skp u :: s)
  UNMARK : ∀ {u s} → Instr (Val u :: Skp u :: s) (Val u :: s)

Code : Shape → Shape → Set
Code = Star Instr
```

Note that the types Instr and Code are no longer mutually recursive and code sequences are truly just plain sequences of simple instructions.

3.5.2 Compiler

As already hinted in Section 3.5 on page 51, the compiler remains mostly the same as the one in Section 3.4.2 on page 47. The only change takes place in the clause for the constructor Catch.

```
compile : ∀ {u s} → Exp u → Code s (Val u :: s)
compile (Lit x) = ⟦ PUSH x ⟧
compile (Bin op l r) = compile r ⋘ compile l ⋘ ⟦ opInstr op ⟧
compile Throw = ⟦ THROW ⟧
compile (Catch e h) =
    ⟦ MARK ⟧ ⋘ compile e ⋘ ⟦ HANDLE ⟧ ⋘ compile h ⋘ ⟦ UNMARK ⟧
```

Clearly seen here, no instruction except PUSH takes any argument whatsoever, and the resulting code is completely linear.

3.5.3 Execution

This section describes what the machine state looks like and how it changes as instructions are executed.

Machine state

As usual, we will need some auxiliary functions so that we can express the shapes of stacks in type declarations.

The function unwindShape is an old hat: it calculates the shape of the stack on top of which the top-most handler would be run if an exception occurred now.

```
-- Unwind the shape up to just below the n-th handle-mark from the top.
unwindShape : Shape → ℕ → Shape
unwindShape (Han _ :: xs) zero = xs
unwindShape (Han _ :: xs) (suc n) = unwindShape xs n
unwindShape (Skp _ :: xs) n = unwindShape xs n
unwindShape (Val _ :: xs) n = unwindShape xs n
unwindShape [] _ = []
```

In addition, we also define the function skipShape that calculates the shape of the stack with which execution will continue after skipping the top-most handler, if no exception occurs.

```
-- Shape of the stack after skipping the n-th skip-mark from the top.
skipShape : Shape → ℕ → Shape
skipShape (Han _ :: xs) n = skipShape xs n
skipShape (Skp u :: xs) zero = Val u :: xs
skipShape (Skp _ :: xs) (suc n) = skipShape xs n
skipShape (Val _ :: xs) n = skipShape xs n
skipShape [] _ = []
```

Simply put, the function unwindShape unwinds the shape just below the n-th "handler placeholder"; the function skipShape unwinds the shape just below the n-th "skip placeholder" (preserving the Val u on top that comes from a successful evaluation of either the main expression code or the handler code).

Now we are ready to define the machine state. The state is modeled by a data type with three alternatives/constructors: the regular state, the exception-handling state, and the new handler-skipping state.

```
data State (s : Shape) : Set where
  ✓[_] : Stack s → Shape s
  ×[_,_] : (n : ℕ) → Stack (unwindShape s n) → State s
  ![_,_] : (n : ℕ) → Stack (skipShape s n) → State s
```

The two alternative constructors are very similar to each other, differing only in how the resulting stack shape is calculated from the current one.

STATE TRANSITIONS

The core of execution is given by the function execInstr (Figure 3.4 on the facing page), that describes effects of single instructions on the state of the machine.

Note that, like the instruction/code type declarations in Section 3.5.1 on page 52, the functions execCode and execInstr are no longer mutually recursive, and, as always, the function execCode is a simple left fold over instructions.

This concludes the operational semantics of instructions of the low-level virtual machine.

3.5.4 Correctness

The general schema of the proof is identical to that in Section 3.4.5. The proof differs only in minor details, despite the fact that we have three machine states now. Basically, all functions performing case analysis get several new clauses dealing with the new handler-skipping machine state – and that's all.

The detailed proof can be found in the accompanying code.

```
execInstr : ∀ {s t} → Instr s t → State s → State t
-- Normal operation
execInstr (PUSH x) ✓[          st ] = ✓[ x :-: st ]
execInstr ADD      ✓[ x :-: y :-: st ] = ✓[ (x + y) :-: st ]
execInstr MARK     ✓[          st ] = ✓[ han:-: skp:-: st ]
execInstr THROW    ✓[          st ] = ×[ zero , unwindStack st zero ]
execInstr UNMARK   ✓[ x :-: skp u :-: st ] = ✓[ x :-: st ]
-- Exception handling: trivial
execInstr THROW    ×[     n , st ] = ×[     n , st ]
execInstr (PUSH x) ×[     n , st ] = ×[     n , st ]
execInstr ADD      ×[     n , st ] = ×[     n , st ]
execInstr UNMARK   ×[     n , st ] = ×[     n , st ]
execInstr MARK     ×[     n , st ] = ×[ suc n , st ]
execInstr HANDLE   ×[ suc n , st ] = ×[     n , st ]
-- Forward jump: trivial
execInstr THROW    ![ n , st ] = ![ n , st ]
execInstr (PUSH x) ![ n , st ] = ![ n , st ]
execInstr ADD      ![ n , st ] = ![ n , st ]
execInstr HANDLE   ![ n , st ] = ![ n , st ]
execInstr MARK     ![     n , st ] = ![ suc n , st ]
execInstr UNMARK   ![ suc n , st ] = ![     n , st ]
execInstr UNMARK   ![ zero  , st ] = ✓[ st ]
-- Exception handling: run the handler on exception
execInstr HANDLE   ×[ zero , st ] = ✓[ st ]
-- Exception handling: no exception, skip the handler, keep the stack
execInstr HANDLE   ✓[ x :-: han:-: skp:-: st ] = ![ zero , x :-: st ]

execCode : ∀ {s t} → Code s t → State s → State t
execCode ε st = st
execCode (i ◁ is) st = execCode is (execInstr i st)
```

Figure 3.4: Linearized code: execution functions

3.6 Adding types and binary operators

To demonstrate that having done all this work, adding new types and operators is easy and straightforward, we extend our high-level language with the Boolean type, along with two new binary operators: the Leq operator that compares natural numbers, and the And operator calculating conjunction of two Boolean values.

The changes are simple, straightforward, and mechanical. They mostly involve adding new cases in pattern matches in various places. The upside is that, where relevant, the number of cases grows linearly in the number of operators. The downside is that the constant factor in one of the lemmas belonging to the correctness proof is as much as 12 (trivial) clauses per binary operator.

The corresponding Agda code can be found in the attached development, in the subdirectory sec3.6-more-ops.

3.6.1 High-level language

TYPE UNIVERSE

We start by extending the type universe with the new Boolean type, along with its interpretation in Agda types.

```
data U : Set where
  nat : U
  bool : U
```

In the above type declaration, we just added the bool constructor. Now we need to extend the function el with a new clause covering the case for the newly introduced bool constructor.

```
-- open import Data.Bool
data Bool : Set where
  true : Bool
  false : Bool

el : U → Set
el nat = ℕ
el bool = Bool
```

EXPRESSIONS

The type of expressions Exp remains unchanged; we just need to add the two new binary operators to the data type Op as new, appropriately typed, constructors.

```
data Op : U → U → U → Set where
  Plus : Op nat nat nat
  Leq : Op nat nat bool
  And : Op bool bool bool
```

Recall that a constructor of the type `Op p q r` represents a binary operator taking two values of the types `p` and `q`, returning a value of the type `r`.

Denotational semantics

Since we changed only the operator data type, we need to alter just the corresponding denotation function `denOp`.

First, we add two auxiliary functions operating on Agda types, implementing the action of the two new operators.

```
infixl 5 _∧_
_∧_ : Bool → Bool → Bool
false ∧ b = false
true ∧ b = b

infix 5 _≤_
_≤_ : ℕ → ℕ → Bool
zero ≤ n = true
suc m ≤ suc n = m ≤ n
_ ≤ _ = false
```

Now we can use these functions as the denotation of the new operators.

```
denOp : ∀ {u v w} → Op u v w → el u → el v → el w
denOp Plus = _+_
denOp Leq = _≤_
denOp And = _∧_
```

3.6.2 Low-level language

Instructions

Next, we extend the virtual machine with two new instructions representing the two new operations. We do this through adding new constructors to the `Instr` data type, as shown in Figure 3.5 on the following page.

```
data Instr : Shape → Shape → Set where
  PUSH : ∀ {u s} → el u → Instr s (Val u :: s)
  THROW : ∀ {u s} → Instr s (Val u :: s)
  ADD : ∀ {s} → Instr (Val nat :: Val nat :: s) (Val nat :: s)
  MARK : ∀ {u s} → Instr s (Han u :: Skp u :: s)
  HANDLE : ∀ {u s} → Instr (Val u :: Han u :: Skp u :: s) (Skp u :: s)
  UNMARK : ∀ {u s} → Instr (Val u :: Skp u :: s) (Val u :: s)
  -- new instructions
  LEQ : ∀ {s} → Instr (Val nat :: Val nat :: s) (Val bool :: s)
  AND : ∀ {s} → Instr (Val bool :: Val bool :: s) (Val bool :: s)
```

Figure 3.5: Additional operations: data type of instructions

COMPILER

After adding the two instructions, we relate them to the corresponding operators by adding new clauses to the function opInstr. Recall that this function determines what instructions different binary operators compile to.

```
opInstr : ∀ {u v w s} → Op u v w → Instr (Val u :: Val v :: s) (Val w :: s)
opInstr Plus = ADD
opInstr Leq = LEQ
opInstr And = AND
```

EXECUTION

Finally, we need to extend the instruction interpreter to include the two new instructions.

We add a pair of clauses into every section of the function: *normal operation*, *trivial exception handling*, *trivial forward jump*, as shown in Figure 3.6 on the next page.

Note that the only non-trivial addition was the one in the *normal execution* section, which defines the operational semantics of the two new instructions. The remaining two pairs are just obligatory no-ops, consistent with the skipping interpretation of these two modes of execution.

3.6.3 Correctness

The largest chunk of code gets added to the proof of correctness, namely the operator lemma called lemma-op. In this function, 12 cases per binary operator are examined

```
execInstr : ∀ {s t} → Instr s t → State s → State t
-- Normal operation
execInstr (PUSH x ) ✓[           st ] = ✓[ x :-: st ]
execInstr ADD       ✓[ x :-: y :-: st ] = ✓[ (x + y) :-: st ]
execInstr MARK      ✓[           st ] = ✓[ han:-: skp:-: st ]
execInstr THROW     ✓[           st ] = ×[ zero , unwindStack st zero ]
execInstr UNMARK    ✓[ x :-: skp u :-: st ] = ✓[ x :-: st ]
execInstr LEQ       ✓[ x :-: y :-: st ] = ✓[ (x ≤ y) :-: st ]
execInstr AND       ✓[ x :-: y :-: st ] = ✓[ (x ∧ y) :-: st ]
-- Exception handling: trivial
execInstr THROW   ×[     n , st ] = ×[     n , st ]
execInstr (PUSH x) ×[     n , st ] = ×[     n , st ]
execInstr ADD     ×[     n , st ] = ×[     n , st ]
execInstr UNMARK  ×[     n , st ] = ×[     n , st ]
execInstr MARK    ×[     n , st ] = ×[ suc n , st ]
execInstr HANDLE  ×[ suc n , st ] = ×[     n , st ]
execInstr LEQ     ×[     n , st ] = ×[     n , st ]
execInstr AND     ×[     n , st ] = ×[     n , st ]
-- Forward jump: trivial
execInstr THROW   ![ n , st ] = ![ n , st ]
execInstr (PUSH x) ![ n , st ] = ![ n , st ]
execInstr ADD     ![ n , st ] = ![ n , st ]
execInstr HANDLE  ![ n , st ] = ![ n , st ]
execInstr MARK    ![     n , st ] = ![ suc n , st ]
execInstr UNMARK  ![ suc n , st ] = ![     n , st ]
execInstr UNMARK  ![ zero  , st ] = ✓[ st ]
execInstr LEQ     ![ n , st ] = ![ n , st ]
execInstr AND     ![ n , st ] = ![ n , st ]
-- Exception handling: run the handler on exception
execInstr HANDLE  ×[ zero , st ] = ✓[ st ]
-- Exception handling: no exception, skip the handler, keep the stack
execInstr HANDLE  ✓[ x :-: han:-: skp:-: st ] = ![ zero , x :-: st ]
```

Figure 3.6: Additional operations: execution functions

so we end up with 2×24 new trivial[21] clauses. However, the rest of the proof remains completely untouched.

The precise formulation of the proof can be found in the attached code.

3.6.4 Remarks

Thus, our development is extensible in this way quite easily. This wouldn't be possible if, instead of keeping track of the types of elements on the stack, we kept just the size of the stack.

Of course, there are lots of things to improve. Most notably, the large number of clauses in the operator lemma belonging to the proof of correctness might probably be factored into smaller functions, resulting in slower growth of the clause count when adding new instructions.

Possible ways to deal with this issue could be metaprogramming in the spirit of Template Haskell, reflection, or, perhaps most readily available, automation in the form of a tactic language, as found in Coq.

[21] All these proof obligations can be discharged using a simple refl.

Chapter 4

Discussion

4.1 Further work

There are *lots* of things in this development that could be improved and many alternatives to the design choices of ours that could be interesting to explore further.

In Section 3.3.5 on page 44, we mentioned the transactionality of exceptions in pure code. Transforming linear code to the form of "forks" mentioned there is not too hard – although not trivial – and this intermediate representation could provide a reasoning bridge between tree-structured expressions and linear code. The compiler could even emit this intermediate structure first, linearizing it later, which might facilitate proving the part of correspondence between "forks" and linear code. Such an intermediate structure might then be useful in proving termination and correctness of the program.

There are other models that could help proving the desired properties. In Section 3.3.1 on page 37, we introduced the notion of handler frames that correspond to catch-expressions. However, this notion can be generalized to all other types of expressions, yielding *expression frames*: any expression, not only instances of the constructor Catch corresponds to a frame similar to the one in Figure 3.1 on page 37. A suitable model of expression frames could be another intermediate structure that may be useful in proving termination and correctness.

Instead of structural recursion on the code sequence, a decreasing measure could be used. This would enable arbitrary jumping within the code and execution on the virtual machine could be modeled very closely to how real machines work.

The high-level language could be extended by functions or lambdas. Besides higher expressivity of regular code, this would enable exception handlers to inspect

exceptions and behave differently in different cases. This also means that it would make sense to throw values and the syntax of the throw-expressions would be extended to include the value to be thrown. Finally, once values are added to `throw`, type-based selection of handlers could be implemented.

On the other hand, the high-level language could be made to support just one type: numbers. This would bring some simplification to the Agda code – stack shapes would be replaced by simple stack sizes – and perhaps help resolve termination in the cases where we were unable to prove it. Usability of this approach depends on the requirements; if all that is needed are numbers, generalization to other data types makes the code unnecessarily complicated and this simplification would pay off.

In Section 3.6.3 on page 58 we mention that the operator lemma in the proof of correctness must contain as much as 12 (trivial) clauses per operator. A suitable factorization of the lemma into smaller functions might reduce this constant; employing some automation technique might remove it altogether.

Unfortunately, our structurally recursive model of execution cannot include jumps, which are an important part of functionality of real-world machines. The paper by Hutton and Wright [HW04] uses labels to mark locations in code that may be jumped to. Although we cannot jump, it might be useful to add labels and prove that jumping to them is equivalent with our mode-switching execution strategy. This would make translation of our algorithm to real-world implementations easier and more obvious.

4.2 Related work

Besides the paper about *verification* of a compiler of exceptions [HW04], later formalized by Nipkow [Nip04], Hutton and Wright also published a paper that *calculates* a virtual machine from the same high-level language with exceptions via defunctionalization [HW06].

McKinna and Wright created a certified compiler of expressions [MW06] in Epigram. Their compiler does not have exceptions but it features numeric and Boolean types and conditions from the beginning. Their development also makes heavy use of dependent typing, which makes their approach very close to ours in this respect.

Chlipala presents a "verified compiler for an impure functional language" [Chl10], implemented in Coq, trying to assess what ways might be viable to write a real-world certified compiler. Besides other features, the impure functional language also includes exceptions. The compiler is quite elaborate and consists of multiple stages, including an optimization stage. The development makes extensive use of

the Coq tactic language to automate proofs, and PHOAS[1] to deal with binders, in the effort to save as much work as possible when both implementing and extending the compiler. The paper does not discuss execution of the resulting code on a virtual machine and does not deal with termination of the execution function.

Leroy presents a verified realistic compiler of (a substantial subset of) the C programming language, named CompCert [Ler09], also written and verified in Coq. This is a full-featured real-world compiler of a mainstream procedural language to PowerPC machine code, dealing with advanced topics like optimizations and register allocation in a completely certified way, while keeping the performance of the emitted code on par with that of the mainstream C compilers. Being a C compiler, it does not support exceptions.

4.3 Conclusions

Finally, we converged to a solution that meets our objectives: a certified compiler for a language with exceptions, along with a tail-recursive executable interpreter of the corresponding low-level code. While we have diverged from the paper by Hutton and Wright [HW04] in low-level execution-related matters by using different machine modes for instruction skipping instead of jumping, the overall direction of our development happens to match that of the paper, while adding structurality everywhere.

We use the power of dependent types to keep relatively strong invariants; as a consequence, we needn't bother with cases that are impossible, such as executing the instruction ADD on an empty stack. We set up the types so that they rule out all such cases, which would be impossible in languages with weaker type systems, such as Haskell or OCaml.

However, this did not take too much work, when compared to such languages. Type signatures are a bit more elaborate but the code itself is very similar. The whole Agda development is 180 lines of code, and 120 lines of proofs, all inclusive.

When we look at the objectives listed in the Introduction on page 1, we can conclude that they have been met.

- Our program is *runnable*; it is structurally recursive, obviously terminating, and provably correct with respect to the presented specification.

- The code is *readable*. Especially because all recursion is structural, there are no proof terms in the informative part of the development. The small-step operational semantics of individual instructions is given clearly by the tabular form of the function execInstr (Section 3.5.3 on page 54). And, as already

[1] Parametric higher-order abstract syntax [Chl08].

mentioned, also the rest of code looks similarly to what the counterparts in other functional languages would look like.

- The low-level code is simple enough to be *executable by a stack machine*. Except for a small and limited amount of state, we do not use anything more, especially not implicit stacks, nor arbitrarily-sized instructions.

- *Extraction of code* to other languages should also be simple, mostly due to the properties that make the code readable; in particular, due to the absence of of proof terms in the informative/executable part of the development, which follows from structurality of the recursion used, and due to the similarity to what the program would look like in other functional languages.

It has been 45 years since the seminal paper by McCarthy and Painter [MP67] opened up the door into the world of verified compilers by introducing the first formally verified compiler. In the meantime, many things have changed, theory has been developed and machines have become more powerful, able to not even check our proofs but also assist us in writing them. To fully reap the power of the machines in this area, we need to develop efficient methods of formal verification – and while lots of ingenious work has been done in this area, it is still a long and exciting way to go. Let this thesis be a small step towards that goal.

Bibliography

[Agd12a] Agda Wiki authors. The Agda Wiki. http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.Othertutorials, 2012. Retrieved on Jul 26, 2012.

[Agd12b] Agda Wiki authors. The Agda Wiki: Quick guide to editing, type checking and compiling Agda code. http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.QuickGuideToEditingTypeCheckingAndCompilingAgdaCode, 2012. Retrieved on Jul 26, 2012.

[BC05] Ana Bove and Venanzio Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(04):671–708, 2005.

[BD08] Ana Bove and Peter Dybjer. Dependent types at work. In Ana Bove, Luís Soares Barbosa, Alberto Pardo, and Jorge Sousa Pinto, editors, *LerNet ALFA Summer School*, volume 5520 of *Lecture Notes in Computer Science*, pages 57–99. Springer, 2008.

[Chl] Adam Chlipala. Library StackMachine. http://adam.chlipala.net/cpdt/html/StackMachine.html. Accessed on July 10, 2012.

[Chl08] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 143–156, New York, NY, USA, 2008. ACM.

[Chl10] Adam J. Chlipala. A verified compiler for an impure functional language. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 93–106, 2010.

[Con08] Conor McBride. Parameterized type families. https://lists.chalmers.se/pipermail/agda/2008/000594.html, 2008. Agda mailing list, post from Oct 26 2008 at 23:14:37 CET.

[Dyb97] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 1997.

[Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[How80] William A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980.

[Hur95] Antonius J. C. Hurkens. A Simplification of Girard's Paradox. In *TLCA*, pages 266–278, 1995.

[HW04] Graham Hutton and Joel Wright. Compiling Exceptions Correctly. In *Proceedings of the 7th International Conference on Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, Stirling, Scotland, July 2004. Springer.

[HW06] Graham Hutton and Joel Wright. Calculating an Exceptional Machine. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programming volume 5*. Intellect, February 2006. Selected papers from the Fifth Symposium on Trends in Functional Programming, Munich, November 2004.

[LDF+11] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 3.12. http://caml.inria.fr/pub/docs/manual-ocaml/, 2011. Accessed on July 10, 2012.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[LL12] Kenneth C. Louden and Kenneth A. Lambert. *Programming Languages: Principles and Practice, 3rd edition*. Cengage Learning, 2012.

[MP67] John Mccarthy and James Painter. Correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 33–41. American Mathematical Society, 1967.

[MW06] James Mckinna and Joel Wright. A type-correct, stack-safe, provably correct, expression compiler. In *Submitted to the Journal of Functional Programming*, 2006.

[Nip04] Tobias Nipkow. Compiling exceptions correctly. *Archive of Formal Proofs*, 2004.

[Nor08] Ulf Norell. Dependently typed programming in agda. In *Lecture Notes from the Summer School in Advanced Functional Programming*, 2008.

[RD11] Guido van Rossum and Fred L. Drake Jr. *The Python Language Reference Manual (version 3.2)*. Network Theory Ltd., 2011.

[The12] The GHC Team. *The Glorious Glasgow Haskell Compilation System, User's guide, Version 7.4.2*, 2012. Available at `http://www.haskell.org/ghc/docs/7.4.2/users_guide.pdf`, accessed on Aug 1, 2012.

List of Figures